JPL PUBLICATION 80-73

Fault-Tolerant Computer Study

Final Report

David A. Rennels
Algirdas A. Avizienis
Milos D. Ercegovac

February 1, 1981

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL PUBLICATION 80-73

Fault-Tolerant Computer Study

Final Report

David A. Rennels
Algirdas A. Avizienis
Milos D. Ercegovac

February 1, 1981

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the Naval Ocean Systems Center, San Diego, California, through an agreement with NASA.

ABSTRACT

This report describes a set of building-block circuits which can be used with commercially available microprocessors and memories to implement fault-tolerant distributed computer systems. Each building-block circuit is intended for VLSI implementation as a single chip. Several building blocks and associated processor and memory chips form a self-checking computer module with self-contained input output and interfaces to redundant communications buses. Fault tolerance is achieved by connecting self-checking computer modules into a redundant network in which backup buses and computer modules are provided to circumvent failures.

Included in the report is a discussion of the requirements and design methodology which led to the definition of the building-block circuits. This is followed by a set of logic designs for three of the building blocks. These are designs which are being used to construct a laboratory breadboard of a self-checking computer module. The logic designs will be modified and improved as the breadboard is debugged and tested. Further refined designs will become available when the breadboard is completed and tested and again, hopefully, when the VLSI devices are fabricated.

ACKNOWLEDGMENT

This study was initiated by the Naval Ocean Systems Center, Code 923, and represents a facet of a block-funded program entitled Integrated Circuit Technology, sponsored by the Naval Electronics Systems Command, Technology Division. The work was performed by agreement with NASA under Contract NAS7-100 at the Jet Propulsion Laboratory of the California Institute of Technology. This program is continuing under NASA sponsorship, and related system studies are being conducted at the University of California, Los Angeles under sponsorship of the Office of Naval Research.

A special acknowledgment is due to Reeve Peterson of NOSC for his continued support and encouragement of this effort. We are also indebted to Dick Urban and Ed Holland of NOSC for their guidance and support.

For the continuing effort, which involves the detailed design and implementation of an engineering model of this work, an acknowledgment is owed to Lee Holcomb of the NASA Office of Aeronautics and Space Technology for his support.

An additional acknowledgment is due Jim Bryden of JPL whose help was invaluable in carrying out this study and bringing this final report to parturition.

CONTENTS

Figures

Tables

SECTION 1
SUMMARY AND OVERVIEW

Over the last decade, the methodology of fault-tolerant computing has been developed to increase the reliability of computer systems. Fault-tolerant computers have been designed to contain redundant circuits and, when hardware faults occur, they utilize the redundant circuits to continue correct computation. By and large, these have all been customer-designed computer systems [AVIS 77].

This study was undertaken as part of the NOSC Very-Large-Scale-Integrated-Circuit Technology Program to define VLSI building-block circuits which can be used with commercially available microprocessors and memories to implement fault-tolerant computer systems. This approach is taken with the view that a wide range of government requirements can be satisfied with commercially developed processors. Thus, the direction of this study is to define the supporting circuits necessary to utilize existing processors in fault-tolerant configurations.

The principal result is a determination that a small number of building-block circuits can be developed which will allow construction of both centralized and distributed (multi-computer) computer configurations which are fault tolerant. These building blocks consist of (1) an Error Detecting and Correcting Memory Interface Circuit, (2) a CORE Processor Checker and Fault-Handling Circuit, (3) a Self-Checking Programmable Bus-Interface Circuit, and (4) several I/O circuits to perform voting, error checking, and short isolation. The design of the first three building blocks for a feasibility breadboard are described in this report, along with the rationale behind their selection.

1.1 SYSTEM REQUIREMENTS

Reliability is a continuing problem in complex military systems. The cost of unexpected failures shows up in many ways, including reduced operational readiness, and the large number of personnel involved in maintenance. Dollar costs are usually difficult to quantify

because system procurement and costs of ownership are usually parcelled into various areas of responsibility. It can be said, however, that costs of ownership often exceed procurement costs in a large number of major systems.

By increasing testability, maintainability and, in some cases, providing automated redundancy management in the early stages of a system design, it is expected that life-cycle costs can be reduced. This viewpoint advocates moderately increasing initial hardware costs to achieve improved reliability and reduced maintenance during a system's operational lifetime.

The computers within a system provide the starting point for automated maintenance. If computer reliability is assured, the computers can be used for (1) subsystem testing and failure diagnosis, (2) automatically replacing failed subsystems with spare parts, or (3) where no backup spares are available, modifying on-board processing to account for the degraded subsystem state. Stated another way, the computer becomes an automated repairman.

A second area of requirements for fault-tolerant computing occurs when the cost of computer failure becomes clearly unacceptable. Digital flight control of low-flying aircraft is a dramatic example. Although the number of applications of this type is relatively low, they may be expected to increase as the computer is relied upon more heavily.

1.2 BUILDING-BLOCK COMPUTER REQUIREMENTS

The user of a fault-tolerant building-block computer (FTBBC) system should be allowed to specify a maintenance interval and the reliability required over that interval. This has two major implications. First, the FTBBC configurations must allow the modular addition of redundant elements so that the same design, with differing numbers of spares, can economically satisfy both short- and long-life requirements. Secondly, the fault detection and recovery mechanisms of the FTBBC must be nearly perfect. Previous modeling studies have shown that "coverage," (the conditional probability that the system can implement recovery,

given that a fault occurs) must approach 100% for long-term reliability, whether or not a fault-tolerant system is periodically maintained [BOUR 69].

In order to be effective, a fault-tolerant computer must be designed to recover from a comprehensive set of faults, i.e., all the faults that can be reasonably expected to occur. We have attempted to protect against stuck-at faults on a single chip, most massive failures in a single chip or module, and most transient faults which create errors but which are of short duration. We do not expect unrelated hard faults to occur in different modules simultaneously.

The FTBBC architecture must be amenable to easy maintenance. Plug-in replacement modules should require a minimum of contact pins and should not require connectors at high-bandwidth, noise-sensitive points in the computer. Similarly, the computer should be capable of identifying, during routine maintenance, those modules which must be replaced.

The architecture of the building blocks should be capable of supporting a wide variety of processor and memory chips, i.e., the building block designs should not depend upon the peculiar I/O characteristics of any given processor. By initiating all control and I/O functions with out-of-range memory addresses (memory-mapped I/O), this processor independence can be achieved.

For the building-block computers to find wide application they should be consistent with military standardization programs. Thus, external bus interface circuits in the building block architecture use MIL-STD 1553A.

1.3 DESIGN APPROACH

After a study of alternative approaches to the design of building-block-implemented, fault-tolerant computing systems, the following architecture was selected. The building-block circuits being developed are used to assemble commercially available microprocessors and memories into Self-Checking Computer Modules (SCCM), as

shown in Figure 1-1. Each SCCM is a small computer with the unusual property that its hardware is capable of detecting a wide variety of internal faults concurrent with normal (user) program execution. It can be connected (through a redundant external busing system), together with other SCCMs into a redundant network, in which backup SCCMs are provided to take over for a computer (SCCM) which has failed.

As shown in Figure 1-1, three of the building blocks interface (1) local memory, (2) the external busing system, and (3) local I/O to the processor. These interface building blocks are responsible for detecting faults in the circuits that they interface to the SCCM's processor, and faults in their own internal logic. They send fault indicator signals to the Core Building Block (Core-BB) if such a fault is detected.

The Core Building Block compares the outputs of two CPUs performing identical computations to detect (but not isolate) CPU faults, and it receives the fault signals from the other building blocks. It also checks error-detecting codes which are used to detect errors on the internal busses of the SCCM. The Core is responsible for disabling the SCCM upon detecting a fault anywhere within it. (An optional program rollback may be attempted to recover from some transient faults locally.)

Although the primary means of fault recovery is to use backup SCCMs to replace a SCCM which has failed, it is possible to correct some of the most likely faults in a failed SCCM (by an internal reconfiguration) and reuse it. A SCCM can be reconfigured to recover from at least two local memory faults through use of two spare-bit planes. Redundant external Bus Interface Building Blocks (BIBB) allow communication through alternate buses if a bus interface should fail, and redundant I/O Building Blocks can be used within a SCCM. (A design augmentation currently under consideration, allows one of the two CPUs to be discarded when a disagreement occurs, and computation to continue with only one. This is for non-critical applications since CPU fault detection is no longer available with only one machine.)

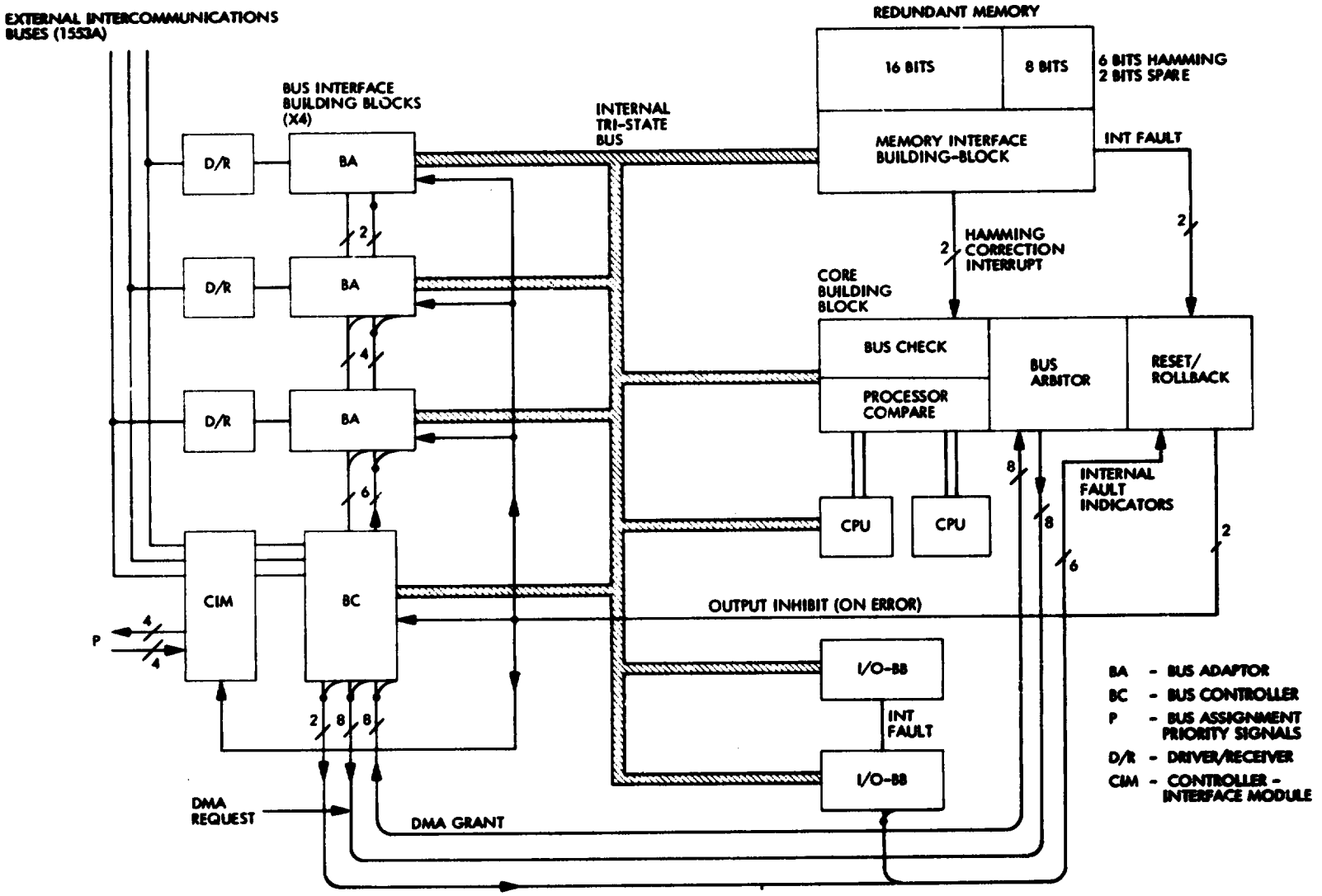


Figure 1-1. The Self-Checking Computer Module (SCCM)

1.4 THE BUILDING-BLOCK CIRCUITS

The building-block circuits are briefly described in the following paragraphs.

1.4.1 The Memory-Interface Building Block (MIBB)

This circuit interfaces a set of commercial memory chips to the local bus within a SCCM. It is capable of detecting single faults within the memory, effecting replacement of up to two faulty bit planes with spares, and correcting single bit errors using a (SEC/DED) Hamming code. It generates and checks parity codes to protect information transfer on the SCCM internal bus. Special checking circuits are employed in the MIBB to detect faults in the memory and within the MIBB, and fault signals are sent to the Core.

1.4.2 The Core Building Block (Core-BB)

This circuit provides a continuous comparison between two processors that run synchronously to detect processor faults. It also includes parity generation and checking circuits to interface the processor with the SCCM local bus and to detect faults on that bus. Internal bus allocation (arbitration) is provided between the CPU and competing DMA channels in the other building blocks. Also, the Core is responsible for disabling its host SCCM in the presence of faults and, optionally, attempting rollback/restart procedures. The Core, like all other building blocks, contains internal checking circuitry to detect faults within its own internal logic.

1.4.3 The Bus-Interface Building Block (BIBB)

This circuit can be microprogrammed to perform the functions of either a controller or terminal (adaptor) to an external 1553A bus. Several BIBBs can be used within an SCCM to provide communications over several redundant external buses.

The BIBB provides the hardware interface between an external bus and the internal bus of its host SCCM. Internal fault-detecting circuitry is provided within the BIBB, and the parity and status

messages employed in 1553A are used to verify proper message transmission and reception.

1.4.4 I/O Building Block (IOBB)

A discussion is included later in this report on the various circuits required to provide fault-detection and redundancy in the interfaces between an SCCM and its associated peripheral devices.

1.5 SCCM PROPERTIES

A "typical" SCCM would consist of the following integrated circuits: 32 commercial RAM chips, 2 commercial microprocessors, 1 MIBB, 1 Core, 3 BIBBs, two IOBBs, and several additional MSI circuits. A previous report has indicated that its characteristics would approximate those listed below if the building blocks were implemented as VLSI devices. (RENN 78a)

Power	8W
Weight	1.4 lb*
Volume	23 $in.^3$*
Cost	$13,600*

*Not including power supply.

The cost represents high reliability production, (e.g., MIL-STD 883B) and could be greatly reduced in large quantities. Figure 1-2 is an estimate of the reliability of a single SCCM, a SCCM backed up by a standby spare, and, for comparison purposes, a non-redundant computer made with similar technology. A simple combinational model was used (see RENN 78a) and it was assumed that a 10,000-gate VLSI device has a failure rate of one failure per million hours. An SCCM costs approximately 50% more in power, weight, volume, and dollars than an equivalent non-redundant machine; but since it can tolerate internal memory faults, its inherent reliability is 2-3 times greater (over the period being modeled). A pair of SCCMs can provide fully fault-tolerant operation with very much improved reliability.

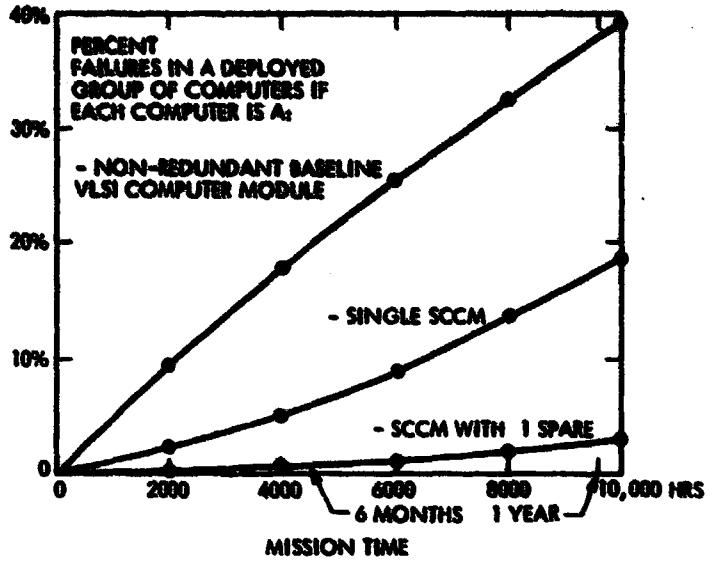


Figure 1-2. Reliability Improvement Using SCCMs

1.6 THE DISTRIBUTED COMPUTER (SCCM) ARCHITECTURE

An architecture has been selected for implementing fault-tolerant distributed computing networks made up of SCCMs. The selected architecture consists of a number of computers (SCCMs) performing separate tasks, and which are connected by a redundant multiple bus structure, as shown in Figure 1-3.

There are two classes of SCCMs used within this network, designated Terminal Modules and High-Level modules. Each Terminal Module is embedded within a particular subsystem and performs local control and data gathering tasks. The High-Level computer modules control the functioning of various terminal modules by controlling an intercommunications bus. Using the bus, a High-Level SCCM can move data directly into or out of memories of other computers and thus broadcast commands or gather data for its various processing functions.

In this configuration, several techniques are employed to achieve fault tolerance. First, all of the computers are self-checking (SCCMs) and are designed to detect their own internal faults.

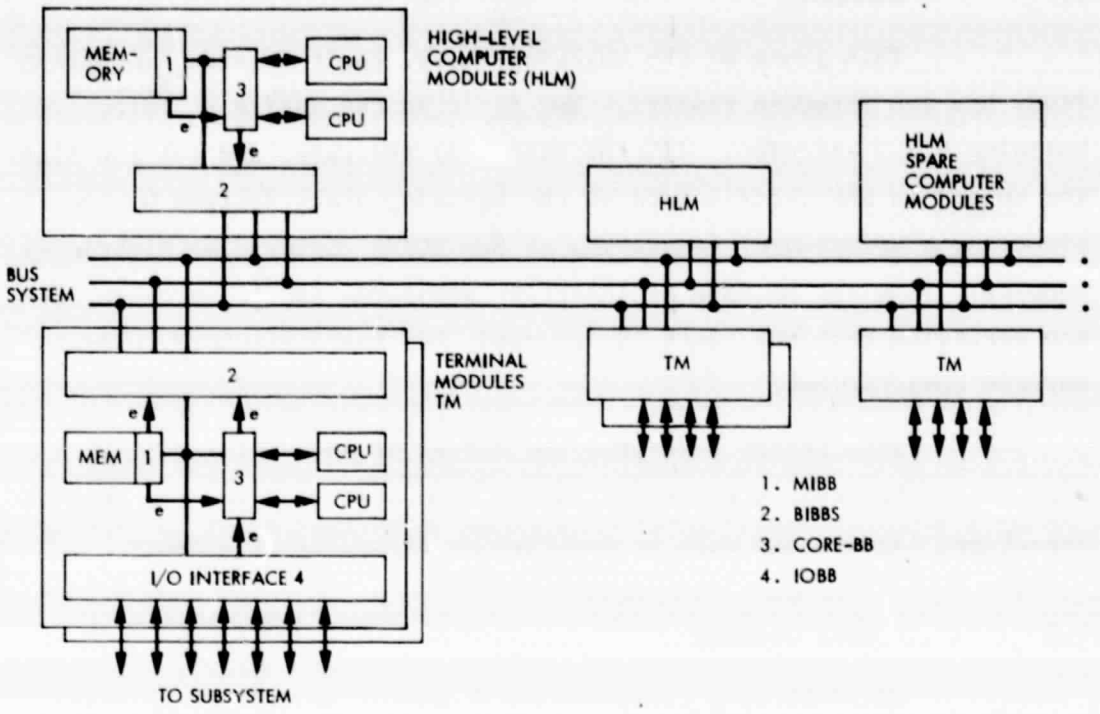


Figure 1-3. Distributed Standby Redundant Architecture

Secondly, backup spares are employed to replace faulty computer modules. In the case of High-Level modules, spares are non-dedicated. A faulty module disables its own bus control function. Spare modules are programmed to detect the resulting lack of activity and take over the ongoing computations. For a Terminal Module, a failure is indicated through the bus system (by polling), and a High-Level Module effects its replacement by activating a dedicated backup spare module.

Thirdly, a highly redundant bus system is employed so that a faulty bus may be replaced by a spare. In the case of single faulty terminals, individual messages may be rerouted over different buses. Automatic status messages are employed in the bus format to verify proper transmission and reception of messages.

A more detailed description of this architecture can be found in RENN 78b.

1.7 SUMMARY

This phase of the building-block, Fault-Tolerant Computing Study has two intended results. The first is the design of three building block circuits: (1) the MIBB, (2) the Core, and (3) the BIBB. The second is the verification of the building-block designs by constructing a breadboard, consisting of two SCCMs employed as high-level modules. This can be done by injecting simulated faults into one SCCM and verifying that the fault is detected, and the other SCCM recovers correct computations.

This report describes the design of the building-block circuits. The designs presented herein have been used for the initial breadboard layout, and will be modified as debugging progresses.

1.8 REPORT OUTLINE

The following two sections (2 and 3) provide background material on the methodology of fault tolerance, and the specific assumptions on technology and application requirements which led to the selection of the building-block SCCM architecture described in this report. The reader who is interested primarily in design details can skip to Section 4, which provides more detailed descriptions of the individual building-block circuits.

SECTION 2

THE CONCEPTS OF FAULT-TOLERANT COMPUTING

The purpose of this section is twofold:

(1) to provide the overall context of fault-tolerant computing as a discipline of computer science and engineering within which the specific results of this study are to be interpreted; and

(2) to supply a self-contained complete introduction to fault-tolerant computer systems for readers who have not encountered this aspect of computer system design in the past.

A fault is an abnormal condition that appears during the operation of an information processing system. Its manifestation may cause a departure from the expected behavior and force the system into an undesirable (error) state or sequence of states. The arrival at an error state, in turn, leads to a partial or complete failure of the system to execute the specified function, unless provisions exist to cause a return to the expected behavior. Causes of faults are either adverse natural phenomena or human mistakes. Because of their disruptive effect on system operation, the avoidance and/or tolerance of faults are major problem areas in contemporary information-processing activities, including the design, analysis, management, and use of information systems

The word "fault" in the subsequent discussion means "an abnormal condition of hardware, programs, or data that may cause a deviation of the information-processing behavior of some part of the given system from the expected sequence," and "system" comprises all hardware elements, programs and microprograms, input signals, stored information, inter-system communication, and man-machine interaction functions. All these parts of the system have to be considered because in practice they all are affected by faults. As a consequence, the fault problem transcends the traditional "hardware-software" applications boundaries and becomes a global problem of information processing.

The word "expected" is preferred to the word "correct" in the description of fault-free behavior because the question of correct behavior, as it has been specified by the originator or user of the system, exceeds the scope of fault-tolerance considerations. For example, the choice of an unsuitable algorithm by the user will lead to expected behavior that is not correct with respect to the user's ultimate goal.

The various types of faults that are encountered during system operation fall into two fundamentally distinct classes: physical faults and man-made faults. Physical faults are faults caused by adverse natural phenomena, such as failures of hardware components, and physical interference originating in the environment. Man-made faults are faults that result from human mistakes, including less than perfect specification, design, production (assembly), and man/machine interaction.

Fault-tolerance is a property of the entire system that allows it to continue the expected behavior regardless of the appearance of certain (explicitly specified) classes of faults (physical, man-made or both) that would otherwise force the system into an error state. The most commonly accepted notion of fault-tolerance refers to physical faults only. The inclusion of man-made faults is a recent generalization that offers a major challenge to investigators and designers of information processing systems.

A complete discussion of fault-tolerance must deal with its three fundamental aspects:

(1) The pathology of faults, including study of their causes, classification according to their immediate manifestations, and characterization according to the symptoms (errors) observable in system behavior.

(2) The implementation of tolerance, encompassing the three basic functions of masking, detection, and recovery.

(3) **The modeling, analysis, and evaluation (measurement) of fault-tolerance by means of mathematical techniques, simulation, and experimentation with implemented systems.**

The goals of this section are: (a) to present a unified view of the many aspects of fault-tolerance; (b) to identify some obstacles that remain to be overcome; and (c) to discuss the prospects for future advances in this field. Fault-tolerance with respect to both physical and man-made faults is considered, with emphasis on the more developed field of tolerating physical faults. The current state-of-the-art in the design and application of fault-tolerant systems is illustrated by examples of existing systems and innovative proposals.

The viewpoint presented here is that the purpose of fault-tolerance is to provide the means for the idealized (fault-free) abstract logical structure of a computing system to function successfully while embodied in its fault-susceptible implementation. Consequently, fault-tolerance attains full significance only when it is incorporated and utilized as an integral function of an information processing system. Outside of this system context, it remains, at best, a potentially applicable exercise for a researcher, and at worst, a tool to support naive or irresponsible promises of near-perfect operation.

2.1 APPROACHES TO THE FAULT PROBLEM

While conceptually the digital computer is a logical system for the storage and manipulation of symbols, in practice it is implemented using physical components and exists in an environment in which it is affected by various natural phenomena. Some phenomena, such as physical changes in the components and adverse effects of the environment, disrupt the operation as it is specified by the designers and programmers and lead to deviations from the expected behavior. These deviations have variously been called failures, faults, errors, intermittents, glitches, crashes, etc. They occur because we attempt to

carry out abstract symbol manipulation operations in a physical world which offers less than perfect components and less than completely benign environments.

The problems of avoiding these phenomena, and of recovering from their effects after they have occurred, have been of interest to the entire community of computer theorists, designers, builders, analysts, and users ever since the first calculating devices were devised. The first pioneers who attempted to implement their ideas were simply overwhelmed by the adversity of the physical world, such as in the case of Babbage's Calculating Engine.

The invention and refinement of electromagnetic relays, vacuum tubes, delay-line and cathode-ray tube storage, paper tape, and punched cards finally made machine computing feasible in the 1940's. However, the history of the early days of machine computing is filled with accounts of the continuing struggle against the imperfections of components and hostility of environments. Ingenious defenses against faults, such as duplicate units, error-detecting codes, etc., are found in most early digital computers. [IRE 53], [EJCC 53].

The advent of the transistor and the magnetic-core storage element in the 1950's brought about a major increase in component reliability and at least temporarily relegated the concern with system reliability into the hands of component experts, and away from the main concerns of system designers and users.

The problem of reliability reappeared as a major issue again in the early 1960's when the applications of computers expanded into the areas of space exploration, real-time system control, and especially manned space-flight, in which the lives of the crew literally depended on successful computer operation.

The reliability of components has continued to improve since that time. However, the expanding range of applications and the growing complexity of systems has kept the reliability problem in the foreground and has led to the evolution of the concept of fault-tolerant computing, which is the designer's and the programmer's method to provide reliable computer operation while using less than perfect components

in less than ideal environments [AVIZ 75a]. The major part of this section considers the tolerance of physical faults; the issue of man-made faults is addressed in Section 2.5

2.1.1 Tolerance and Avoidance: Complementary Approaches to the Fault Problem

A look at computers of the present and of the immediate past shows that many systems have either very few fault-tolerance features, or none at all. In these cases, reliability with respect to physical faults is sought by means of the fault-avoidance approach (also called "fault-intolerance" in some papers) in which the reliability of computing is assured by *a priori* elimination of the causes of faults. The elimination takes place before regular use begins, and the resources that are allocated to attain reliability are spent on perfecting the system prior to its field use. Redundancy is not employed, and all parts of the system must function correctly at all times. Since in practice it has not been possible to assure the complete *a priori* elimination of all causes of faults, the goal of fault-avoidance is to reduce the unreliability (expressed as the probability of system failure before the end of a specified time interval) of the system to an acceptably low value. To supplement this approach, manual maintenance procedures are devised which return the system to an operating condition after a failure. The cost of providing maintenance personnel and the cost of the disruption and delay of computing also are parts of the overall cost of using the fault-avoidance approach. The procedures which have led to the attainment of reliable systems using this approach are:

(1) Acquisition of the most reliable components and their testing under various conditions within the given cost and performance constraints.

(2) Use of thoroughly refined techniques for the interconnection of components and assembly of subsystems.

(3) Packaging and shielding of the hardware to screen out expected forms of external interference.

(4) Carrying out of comprehensive testing of the complete system prior to its use.

Once the design has been completed, a quantitative prediction of system reliability is made using known or predicted failure rates for the components and interconnections. In a "purely" fault-avoiding (i.e., nonredundant) design, the probability of fault-free hardware operation is equated to the probability of correct program execution. Such a design is characterized by the decision to invest all the reliability resources into high-reliability components and refinement of assembly, packaging, and testing techniques. Occasional system failures are accepted as a necessary evil, and manual maintenance is provided for their correction. To facilitate maintenance, some built-in error detection, diagnosis, and retry techniques are provided. This is the most common current practice in computer system design; the trend is toward an increasing number of built-in aids for the maintenance engineer.

The traditional fault-avoidance approach of diagnosis-aided manual repair, however, has proved to be an insufficient solution in many cases because of at least three reasons: the unacceptability of the delays and interruptions of real-time programs (air traffic control, process control, etc.) caused by manual repair action; the inaccessibility of some systems (space, undersea, etc.) to manual repair; and the unacceptably high cost of lost time due to manual maintenance in many installations. The direct dependence of human lives on some computer-controlled operations (air traffic control, manned spaceflight, etc.) has added a psychological reason to object to the fault-avoidance approach: although only one system in a million is expected to fail in a given time interval, all users of the entire million systems are subject to the anticipation that they may be involved in this failure.

An alternate approach which alleviates most of the above shortcomings of the traditional fault-avoidance approach is offered by fault-tolerance. In this approach the reliability of computing is assured by the use of protective redundancy. Faults are expected to be present and to cause errors during the computing process, but their effects are automatically counteracted by the redundancy. Reliable

computing is made possible despite certain classes of hardware failures, external interface with computer operation, and perhaps even some man-made faults in hardware and software. Part of the resources allocated to attain reliability are spent on protective redundancy. The redundant parts of the system (both hardware and software) either take part in the computing process or are present in a standby condition, ready to act automatically to preserve its undisrupted continuation. This contrasts with the manual maintenance procedures which are invoked after the computing process has been disrupted, and the system remains "down" for the duration of the maintenance period.

It is evident that the two approaches are complementary and that the resources allocated to attain the required reliability of computing may be divided between fault-tolerance and fault-avoidance. Experience and analysis both indicate that a balanced allocation of resources between the two approaches is most likely to yield the highest reliability of computing. Fault-tolerance does not entirely eliminate the need for reliable components; instead, it offers the option to allocate part of the reliability resources to the inclusion of redundancy. One reason for the use of a fault-tolerant design is to achieve a reliability or availability prediction that cannot be attained by the purely fault-avoiding design. A second reason may be the attainment of a reliability (or availability) prediction that matches the purely fault-avoiding design at a lower overall implementation cost. A third reason is the psychological support to the users who know that provisions have been made to handle faults automatically as a regular part of the computing process. The fault-avoidance approach clearly was the dominant choice in the 1950's and 1960's. In recent years, the fault-tolerance approach has been making significant inroads with respect to physical faults. Its application with respect to man-made faults has remained very limited.

2.1.2 Classes of Physical Faults

Physical faults are caused by three classes of phenomena that affect the hardware of the system during execution of programs. They are permanent failures of hardware components, temporary

malfunctions of components, and external interference with system operation. There are three useful dimensions for the classification of physical faults:

(1) Duration: transient vs. permanent

(2) Extent: local vs. distributed

(3) Value: determinate vs. indeterminate

Transient faults are faults of limited duration, caused either by temporary malfunctions of components or by external interference. The characterization of a transient fault must include a "maximum duration" parameter; faults that last longer will be interpreted as permanent by recovery algorithms. Other characteristics are the arrival model and the duration of transients [AVIZ 75a]. Permanent faults are caused by irreversible failures of components. They are characterized by the failure rate parameter; often two or more failure rates are used for the same components under different conditions such as power-on and power-off states. The following classifications according to extent and according to value are applicable to both transient and permanent faults.

The extent of a fault describes how many logic variables in the hardware are simultaneously affected by the fault which is due to one failure phenomenon. Local (single) faults are those that affect only single logic variables, while distributed (related multiple) faults are those that affect two or more variables, one module, or an entire system. The physical proximity of logic elements in contemporary MSI and LSI circuitry has made distributed faults much more likely than in the discrete component designs of the past. Distributed faults are also caused by external interference and by single failures of some critical elements in a computer system, i.e., clocks, power supplies, switches used for reconfiguration, etc.

The value of a fault is determinate when the logic values affected by the fault assume a constant value ("stuck on 0" or "stuck on 1") throughout its entire duration. The fault is indeterminate when it varies between "0" and "1" throughout the duration of the fault, but not in accord with design specifications. The determinacy of a

fault depends on the failure mechanism. For example, drift of component values or "shorting together" of two signals are likely to cause indeterminate faults.

It is important to note that the description of fault extent and fault value applies at the origin of the fault; that is, at the point at which the failure phenomenon has actually taken place. The fault-caused introduction of one or more incorrect logic values into the computing process often leads to more extensive fault symptoms farther away (in space and/or in time) from the point of failure. At other times, the presence of incorrect logic value is masked by other (correct) logic variables and no symptoms at all appear at more remote points. Confusion and ambiguity are avoided when the term "fault" is restricted to the change in logic variable(s) at the point of the physical hardware failure. The fault-caused changes of logic variables which are observed farther away on the outputs of correctly functioning logic elements will be called "errors." This choice of terms describes the following cause-effect sequence:

(1) The failure, which is a physical phenomenon, causes a fault, which is a change of logic variable(s) at the point of failure.

(2) The fault supplies incorrect input(s) to the computing process and may cause an error to be produced by subsequent operations of failure-free logic circuits.

The number of points that can be observed for the purpose of fault detection is limited because integrated circuits are internally complex, and have relatively few outputs. Digital-logic simulation programs which analyze the behavior of faulty logic circuits and predict the errors that will appear on the outputs (for a given class of faults) are essential tools for the generation of fault-detection tests [SZYG 76]. An illustration of a simulation and analysis program to analyze the behavior of faulty circuits is the Logic Analyzer for Maintenance Planning (LAMP) system [CHAN 74]. In addition, LAMP also performs logic design verification, generates fault-detection tests, evaluates diagnostics, and produces data for trouble-location manuals. LAMP

exemplifies the current trend toward multipurpose simulation systems in digital system design.

2.2 TOLERANCE OF PHYSICAL FAULTS

Fault-tolerance functions in computer systems are not necessary (redundant) as long as faults do not occur, and they can be deleted from a perfectly fault-free system without affecting its performance. In fault-susceptible systems they are implemented by the means of protective redundancy, which becomes effective when faults occur.

The implementation of fault-tolerance may be discussed from two viewpoints: according to the functions being performed, and according to the forms of redundancy that are used to provide these functions. From the functional viewpoint we distinguish three classes of fault-tolerance functions: masking, detection and recovery. Each class contains several distinct approaches to implementation which will be discussed in this section. The other viewpoint distinguishes different forms of protective redundancy. The redundancy techniques have been developed to enable three different forms: hardware (additional components), software (special programs), and time (repetition of operations).

In this discussion, the functional classification is considered to be most suitable for the exposition of implementation techniques. Each function is discussed separately, outlining the redundancy techniques that are available for its implementation.

2.2.1 Fault Masking

The masking function employs redundancy to assure that the effect of a fault is completely contained within a system module. As long as the redundancy is not exhausted, the fault is concealed within the module and no symptoms whatsoever appear on its outputs. When the redundancy is exhausted or overwhelmed by a fault, module failure results. Separate detection and recovery functions are not identifiable when the module is viewed from outside. Because of this, masking has

been called a static redundancy technique [SHOR 68] and has been used in the design of various structures, e.g., airplane frames, bridges, etc., prior to the appearance of digital systems. Masking is also thought to be the form of fault-tolerance used by the nervous systems of living organisms [VONN 56].

A key question in masking is choice of the size of the module within which the masking occurs. The smallest module is a set of individual hardware components (e.g., diodes, relay contacts, connections, etc.). On the other extreme, a module may be as large as an entire computing system, in which case the module terminals are the output devices. Theoretical analyses of masking usually do not specify the module size; it depends on the feasibility of implementation.

In digital systems, masking is usually accomplished by hardware redundancy, i.e., by the replication of hardware elements. The fundamental theoretical analysis of masking is due to von Neumann [VONN 56], and Moore and Shannon [MOOR 56]. Its early appearance can be attributed to the previous use of masking in other disciplines of engineering. The techniques of introducing hardware redundancy have been classified into two categories: static and dynamic [SHOR 68]. The static method implements the masking function, since the redundant components contain the effect of hardware failures within a given hardware module, and the outputs of the module remain unaffected as long as the redundancy is effective. The static technique is applicable against both transient and permanent faults. The redundant replicas of an element are permanently connected and powered; therefore, they provide fault masking instantaneously and automatically. However, if the redundancy is exhausted, or if the fault is not susceptible to masking and causes an error, a delayed recovery is not provided. In practice, we find that two forms of static redundancy have been applied in U.S. space program computers: replication of individual electronic components, and triple modular redundancy (TMR) with voting [CCOP 76]. Several other forms have been studied but were not applied either because of their excessive cost or because they required practically unrealizable special components [SHOR 68].

The use of static hardware redundancy is based on the assumption that failures of the redundant replicas are independent. For this reason, use of static redundancy is difficult to justify within integrated circuit packages, in which many failure phenomena are likely to affect several adjacent components. Other disadvantages include the cost of massive replication (3, 4 or more times the number of original system elements), the need to assume independent failures of the replicas, and the absence of a warning when a redundant module finally fails. Thus, masking is close to fault avoidance: while it may postpone the time of failure, the module still fails suddenly and irrecoverably when its internal redundancy is exhausted.

Regardless of these shortcomings, masking still may find application because of its conceptual simplicity and its instant action, entirely transparent to the user. A promising area of application is in protecting a small "hard core" of a system for which other approaches are extremely costly or altogether impractical. Another area is the application in non-electrical, discrete-component technologies, such as fluidic logic for high-temperature or extreme radiation environments.

2.2.2 Fault Detection

The detection function is the starting point of all fault-tolerance implementations except for these that depend exclusively on masking. The most sophisticated recovery methods are only as good as the fault detection scheme which initiates their operation. For the purpose of this discussion we say that fault detection has taken place at the time instant at which a fault signal becomes available to be used by a recovery algorithm. All subsequent fault-location actions are considered to be part of the recovery algorithm. The existence of a false fault signal is also possible. This is a false alarm that is due to a malfunction of the fault detection scheme itself.

Fault detection is implemented by means of all the hardware, software and repetition (time) methods that generate the initial fault signal. All these methods may be conveniently grouped according to the

time of their application with respect to the normal operation of the system as follows:

(1) Initial testing, which takes place prior to normal use and serves to identify faults hardware elements containing imperfections introduced during the manufacturing or assembly processes.

(2) Concurrent (on-line) detection, which takes place simultaneously with normal operation of the system.

(3) Scheduled (off-line) detection, which takes place when normal operation is temporarily interrupted.

(4) Redundancy testing, which serves to verify that the various forms of protective redundancy are themselves fault-free, and takes place either concurrently or at scheduled intervals.

Initial testing follows the production of individual circuits and serves to eliminate the circuits that contain manufacturing defects [BREU 76]. Computer programs for test generation have become an essential tool to facilitate initial testing [SZYG 76], [CHAN 74]. The great internal complexity and a relatively small number of input/output points in contemporary LSI circuits (e.g., microprocessors, memories, etc.) have made exhaustive logic-level testing, in many cases economically unfeasible. Recent research has emphasized probabilistic approaches [PARK 76] and combined logic and functional testing [MCPH 76]. Initial testing represents a significant part of the total cost of digital circuits and is likely to remain a high-priority research problem for the foreseeable future.

Concurrent (on-line) fault detection during system operation is implemented by means of special hardware or software that operates concurrently with the regular programs of the system. An important advantage or concurrent detection is that recovery can be initiated before fault-caused errors can cause extensive disruption of programs or damage to the data. Hardware methods for concurrent detection have been

used since the first generation of computers. They include error-detecting codes (parity, etc.) [AVIZ 71a], [DOWN 64], duplication and comparison, [DOWN 64] disagreement detectors with majority voters, [ANDE 67] special circuits to monitor certain critical elements (clocks, power supplies, memory write operation circuits, etc.), [DOWN 64] machine status and completion signals, [AVIZ 71a] self-checking logic circuits, [CART 74] and checksumming, timers, and built-in test equipment of various types.

Software methods for concurrent detection either employ the concurrent execution of two (or more) programs, or they consist of special features interwoven with the single program being executed. In the case of two or more identical programs using separate processors and/or multiple storage in separate memories, a comparison is accomplished by a programmed exchange of results [WENS 76] or checksums, [SKLA 76] rather than by hardware comparators. An alternative is to use a dedicated subsystem (e.g., a "maintenance" minicomputer) which executes monitoring programs to observe the operation of the remaining parts of the system. Fault detection features that can be interwoven with a single program include the use of passwords, acknowledgments ("handshakes"), checksumming, reasonableness checks on results, programmed "watchdog" timers, etc. Compared to hardware methods, fault-detection by software is less prompt and more susceptible to disruption by the fault itself. It is used very widely because it can be superimposed relatively easily on an already existing hardware system.

Scheduled (off-line) fault detection is implemented by means of software and requires the interruption of current programs in order to test for the presence of faults. The presence of errors caused by transient faults can be detected by repeating the execution of the same program (or a program segment) and comparing the results. The detection of permanent faults which may have occurred since the last test period requires the running of diagnostic programs or microprograms [BREU 76], [DOWN 64], [RAMA 72]. In principle they are quite similar to the programs for initial testing. The main differences are: time for testing is usually more strictly limited; testing is executed by the system

itself rather than by another computer; and an interconnected assemblage of various circuits must be tested, rather than one circuit at a time. A "bootstrap" approach is very useful, in which a small part of the system is tested first, and then the tested part is used to run further tests on other parts, etc. Microdiagnostics have very good resolution and are especially suitable for this approach [RAMA 72]. Modern systems also frequently contain special hardware features (e.g., test points) which facilitate diagnostics [CART 64]. Although the present discussion deals with use of diagnostics and microdiagnostics for initial fault detection, we must note that they also often serve to locate detected faults to within a replaceable or discardable module as part of the recovery algorithm.

Redundancy testing is a function that is specifically needed by the fault-tolerance features of a system. Its purpose is to verify that these features will be ready to use when a fault occurs. An especially important aspect is to test that various fault signals are ready to act, i.e., that they are not "stuck" in the "no-fault" state. Self-checking logic [CART 74] and periodic schedule tests of fault signals [CONN 72] are suitable here. A second aspect is the checkout of redundant parts of the system (e.g., standby spares, copies used for masking, etc.). While diagnosis programs are suitable for systems with standby spares [AVIZ 71a], the systems with masking are much more difficult to check out, especially those in which masking is at the component level [COOP 76].

2.2.3 Recovery

The recovery algorithm comprises all actions that are initiated by the arrival of a fault signal during normal operation and are concluded by the resumption of normal operation (possibly in a degraded mode), by a systematic shutdown of the system, or by system failure.

The most fundamental difference between various recovery algorithms is whether interaction with a human maintenance operator is or is not required as part of the recovery algorithm. Recovery algorithms that do not require human decision making are automatic; all

other algorithms are manually controlled, although they may contain extensive automatic (programmed) sequences. An automatic recovery algorithm may make use of off-line manual repair which takes place later, as long as resumption of normal operation does not depend on manual intervention. Automatic recovery algorithms are further classifiable (according to the state of the system after recovery has been completed) into three classes: full recovery, degraded recovery, and safe shutdown.

Full recovery means the return of the system (within allowed time limits) to a set of conditions that existed before the fault occurred [AVIZ 71a]. Both the hardware and software possess the same computing capacity as before. Failed hardware modules are replaced by spares. Damaged information (programs and data) are returned to a known good state that existed prior to the fault.

Degraded recovery (often called "graceful degradation," or "failsoft operation") returns the system to a fault-free state, but with a reduced computing capacity [BEUS 69]. This means that some hardware elements have been discarded without replacement, some programs and/or data have been lost, or some functions have taken longer than the allowed time. This approach may be called "partial fault-tolerance," since recovery is not 100% successful with respect to the set of pre-fault conditions. Various "cold start" procedures belong to this category.

Safe shutdown (also called "fail-safe" operation) is the limiting case for degraded recovery. It is carried out when the remaining computing capacity (if any) is below the minimum acceptable threshold. The goals of shutdown are: to avoid damage to remaining stored information and good system elements; to cease interaction with other systems and/or human users in a specified orderly fashion; and to deliver shutdown messages and diagnostic information to designated systems, users, or maintenance specialists.

Full recovery, degraded recovery, and safe shutdown all require certain subsidiary functions which follow fault detection. They are: fault identification and location, error correction in programs

and data, replacement or exclusion of permanently failed elements, and recording of the observations and actions taken thus far. The final step is either a restart of normal operations, or the completion of the shutdown sequence. Both hardware and software techniques have been devised to implement these functions. They are discussed in more detail in the following section.

2.3 FAULT-TOLERANT SYSTEMS

The ultimate proof of the effectiveness of fault-tolerance techniques is found in the performance of existing systems. For the convenience of discussion, we make the distinction between fully fault-tolerant (or self-repairing) and manually-controlled systems with fault-tolerance features. The former complete their recovery actions without the participation of a maintenance specialist, while the latter depend on human decision making as part of the recovery sequence. These decisions may take place at various stages of the sequence, from the initiation of diagnostics to the operation of the switch which disconnects a failed part of the system.

The fully fault-tolerant systems may be further classified according to the availability of external ("off-line") repair. In closed systems repair is not available, and the system inevitably fails after the redundancy resources have been exhausted. Closed systems are usually found in space applications [COOP 76], [AVIZ 71a], [CONN 72]. In repairable systems, failed parts are automatically identified and excluded from further participation in computing. They are then replaced by an off-line repair action. System failures usually occur either because of imperfect fault detection and recovery algorithms, or because of catastrophic faults (i.e., faults that cannot be handled by the recovery procedures that were provided). A less frequent cause of system failure is exhaustion of redundancy, which occurs when faults occur faster than the repair procedure can handle them. Very prominent examples of repairable systems are the several models of the ESS telephone switching systems [DOWN 64], [BEUS 69].

Finally, fault-tolerance systems may be fixed-capacity or degradable. The former are considered failed if a single specified capacity cannot be maintained, while the latter are allowed to go to one or more configurations of lesser capacity before the system is shut down.

2.3.1 Hardware-Controlled Recovery Systems

Another classification of fault-tolerant systems may be based on the implementation of the recovery algorithm. Hardware-controlled systems have dedicated hardware which collects fault indications and initiates recovery, while software-controlled systems depend on special programs to interpret fault indications and to carry out the automatic recovery procedures. The hardware-controlled recovery approach depends on special hardware to carry out fault detection and to initiate the recovery procedures. After the existence of a properly functioning software system has been assured, the completion of recovery is usually transferred to software control. It is evident that further software systems may be superimposed on the hardware-controlled design, leading to a multilevel recovery procedure. A special case of hardware-controlled recovery is found in statically-redundant systems in which faults are masked by redundant hardware, and thus remain totally invisible to the software. Two examples of such systems are the OAO data processor which used component redundancy and the CPU of the SATURN V guidance computer, which used TMR protection [COOP 76], [ANDE 67]. Probably the earliest use of TMR (triplication and voting) is found in the SAPO computer, designed by A. Svoboda in 1950-53 [OBLO 62]. SAPO also possesses several other fault-tolerance features, including duplication, parity checking, and retry. A separate software-controlled recovery system is needed in statically-redundant systems if they are to continue operating after the first fault escapes the masking effect and affects the software.

Dynamically redundant systems with hardware control usually depend on a dedicated hardware module that gathers fault signals and initiates recovery. Different uses of duplexing and hardware-controlled

switchover techniques are found in the memory, power supply, and peripheral units of the SATURN V guidance computer in combination with a TMR-protected serial CPU unit [ANDR 67]. Separate fault-detection and switchover-control units were used for every functional unit. Probably the first operational computer with fully hardware-controlled dynamic redundancy was the experimental JPL-STAR computer [AVIZ 71a]. Intended for self-contained multiyear space missions, this computer employs a special Test-And-Repair-Processor (TARP) module to control recovery and self-repair. Software assistance is invoked only to perform memory copying and to resume normal operation after self-repair. The French MECRA computer is another early experimental design [MAIS 71]. A few other hardware-controlled system designs that have not reached operation have been described in recent literature [AVIZ 75a], [CONN 72]. An interesting recent experiment is the C.vmp multiprocessor, which can operate in a fault-tolerant mode as a TMR configuration of DEC LSI-11 computers [SIEW 77].

The principal advantage of hardware-controlled recovery systems lies in their independence of the operation of any software immediately after the fault has occurred. The recovery process is transferred to software only after its ability to operate has been assured. The relatively late appearance of such systems may be attributed to the need to introduce the recovery module into the design at its inception, thereby requiring an early commitment to the hardware-controlled approach.

2.3.2 Software-Controlled Recovery Systems

The software-controlled recovery systems depend on special programs to initiate the recovery action upon the detection of a fault. Fault signals are obtained by both hardware and software methods; for example, parity checkers, comparators, power-level monitors, watchdog, timers, test programs, reasonableness checks, etc. The main limitation of these systems is the need for the recovery software to remain operational in the presence of faults, since recovery cannot otherwise be initiated. A significant advantage of the software-controlled approach

is that existing "off-the-shelf" hardware system modules may be used to assemble fault-tolerant organizations. These modules contain various forms of hardware fault detection, which usually are supplemented by further software methods. For this reason software-controlled systems appeared earlier and are currently being used in numerous applications requiring high reliability and availability. While every modern operating system incorporates some recovery features, this report is limited to selected illustrations of historically important and advanced systems.

An important early design of the 1950's that had complete duplication and extensive recovery provisions was the SAGE system [EVER 57]. The IBM System/360 architecture contains very complete serviceability provisions for multi-system operation in order to attain high availability, reconfiguration, and failsoft operation [CART 64]. An early example of a multi-system which includes further extensions of the System/360 design is the IBM 9020 multiprocessing system for air traffic control applications [IBM 67]. Noteworthy are the operational error analysis program and the diagnostic monitor of the 9020. An interesting illustration of extensive use of backup storage and dynamic reconfiguration in a general-purpose time-shared system is found in the MIT Multics System [CORB 72]. The Pluribus is a minicomputer/ multiprocessor system (with extensive fault-tolerance provisions), which serves as a switching node in the ARPA Network [KATS 78]. The TANDEM system is a recently announced commercial multiprocessor system with software-controlled fault-tolerance [TAND 76].

Another direction of software-controlled system development is found in aerospace applications. Representative illustrations of this approach are the SIFT design, [WENS 78] the C.S. Draper Laboratory Symmetric Multiprocessor [HOPK 78] and the COPRA system, [MERA 76] all of which are in design and development stages. An already operational four-computer fault-tolerant complex is the U.S. Space Shuttle computer system [COOP 76], [SKLA 76].

One other area of application which requires fault-tolerant operation and very high availability for several years of continuous operation is the control of electronic telephone switching systems.

These systems usually employ manual repair by replacement of a failed part as the last (off-line) step of the recovery procedure, while maintaining normal operation by means of the remaining system modules. A well-documented illustration is found in the Electronic Switching Systems (ESS) of Bell Telephone Laboratories. The ESS designs use several hardware techniques (duplication, matching, error codes, and functional monitors) and special software (check routines, diagnostics, audits), as well as software and hardware emergency procedures when normal recovery action does not succeed [TOYW 78], [BELS 69]. The Plessey System 250 is a fault-tolerant multiprocessor system for switching system control [HAME 72].

2.3.3 Fault-Tolerant Subsystems

Besides the complete systems discussed above, many efforts have been carried out to provide fault-tolerance for functional subsystems, which then can be assembled to form a fault-tolerant system. This is especially true for secondary and mass storage which has been characterized by relatively low reliability in the past. Representative error coding applications include the use of codes for error control in data communications, magnetic tape units, disc files, primary random access storage, and a photo-digital mass store [TANG 69]. Single-error correcting codes are used in the control storage of the No. 1 ESS [DOWN 64], the main and control storage of IBM System/370 computers, and several other semiconductor memory systems. Error correcting codes have proven to be a very effective method for fault-tolerance in the storage medium, and the remaining problems exist in protection of the memory access and readout circuitry. These have been investigated in an experimental design [CART 76].

Recent studies have considered the problem of fault-tolerance in associative memories and processors [PARH 74]. In general, processor fault-tolerance has been provided by duplication and reconfiguration at the system level. Investigations have been conducted on the use of arithmetic error codes to detect errors caused by processor faults

[AVIZ 71b] and an experimental processor has been designed and constructed for the JPL-STAR computer [AVIZ 71a]. Continuing reductions in the cost of processor hardware make further emphasis on duplication or triplication [HOPK 78] very likely, although error-detecting codes remain a convenient method for the identification of the faulty processor in a disagreeing pair. An exception is found in large scientific computers with multiple arithmetic processors, in which replication is not practical, and graceful degradation procedures must be employed [AVIZ 77a]. A potentially very effective approach to error detection in integrated circuits of processors is self-checking logic design [CART 74], [WAKE 74].

2.4 MODELING AND ANALYSIS

The choice of fault-tolerance functions and redundancy techniques needs to be supported by an assessment whether the system possesses the expected fault-tolerance. Insufficiencies of the design may be uncovered, and the design can be refined by changes or additions of various forms of redundancy. There are two approaches to the evaluation of fault-tolerance:

(1) The analytic approach, in which fault-tolerance measures of the system are obtained from a mathematical model of the system.

(2) The experimental approach, in which faults are inserted either into a simulated model of a system, or into a prototype of the actual hardware, and fault-tolerance measures are estimated from statistical data.

The principal quantitative measures of the effectiveness of fault-tolerance are reliability (with respect to permanent faults) and survivability (with respect to transient faults) [AVIZ 75a]. Methods for the prediction of these measures are discussed in this section.

2.4.1 Analytic Modeling: Permanent Faults

A quantitative reliability prediction for a system requires the knowledge of numerical failure rates of the components, which are given in failures/hour and are usually assumed to be constant. If technologies which are under development are to be used in a new design, the failure rates need to be extrapolated or predicted analytically. Different and possibly time-dependent failure rates may apply to some classes of failures, such as those causing distributed faults. The reliability $R(t)$ is a function of the failure rates and is defined as the probability of the survival of the functional capabilities of a set of hardware elements up to the time t, given that all hardware was in a perfect condition at the time $t = 0$. For a non-redundant system and constant failure rates, the reliability is $R(t) = e^{-\lambda t}$, where λ is the sum of the failure rates of all components (system A of Figure 2-1). In this case, all components have to survive up to the time t. Fault-tolerance of the system is attained only if correct program execution is maintained by the surviving hardware; for this reason the survivability with respect to transient faults must also be considered in a complete evaluation.

A very common quantitative measure used to compare two or more different designs has been the MTTF (mean time to failure), defined as $\text{MTTF} = \int_0^\infty R(t)\,dt$. Given the non-redundant system reliability $R(t) = e^{-\lambda t}$, we have $\text{MTTF} = 1/\lambda$ and the comparison of several MTTF's directly compares the failure rates (λ) of the competing systems. When redundancy is introduced, the reliability function $R(t)$ becomes a polynomial in $e^{-\lambda t}$ (e.g., system B in Figure 2-1) and the $R(t)$ curves of systems being compared may have crossover points. In this case, the area under the curve does not indicate which system is better for a given time interval, and the MTTF may become a misleading measure. Two more precise measures of comparison are illustrated in Figure 2-1 and are discussed below.

Given a fixed "mission time" T, for which the highest reliability is desired, the comparison of two systems requires only the values of $R_A(T)$ and $R_B(T)$ in order to select the best system. The

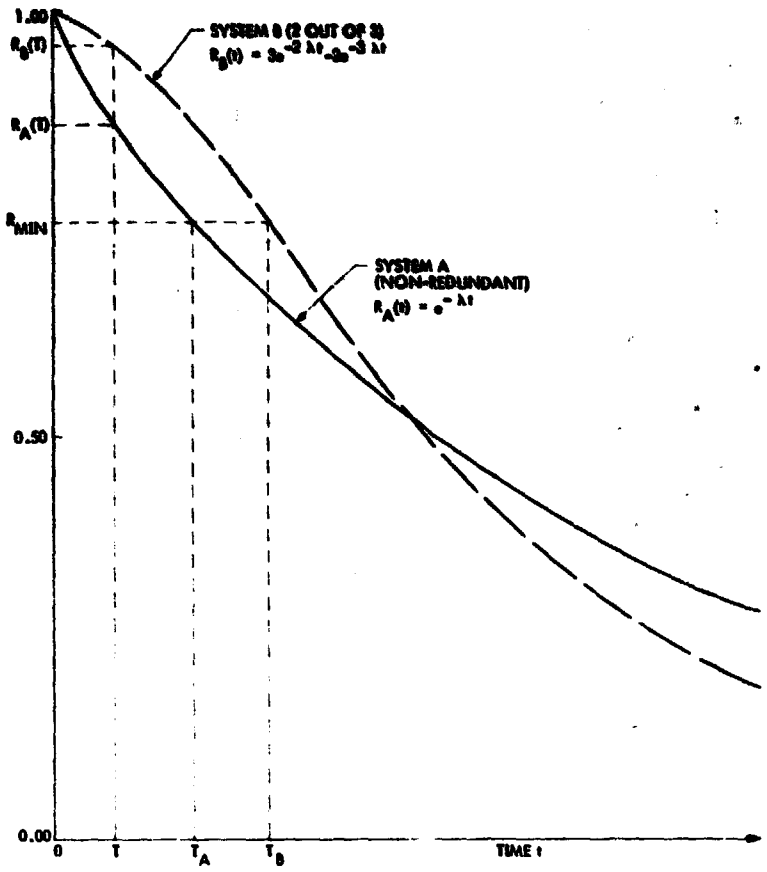


Figure 2-1. System Reliability Predictions.

Reliability Improvement Factor is defined as RIF = $(1 - R_A)/(1 - R_B)$ at the specified mission time T, and it serves as a measure of improvement attained by using the "B" system [ANDE 67]. When a fixed mission time is not specified, the Mission Time Improvement Factor (MTIF) serves as a convenient comparison measure [BOUR 69]. It is defined as MTIF = (T_B/T_A) at R_{MIN}, where R_{MIN} is a specified reliability (e.g., .99 or .90), while T_A and T_B are times at which the system reliabilities $R_A(t)$ and $R_B(t)$, respectively, fall to the value R_{MIN}.

We observe that reliability modeling remains useful even if specific numerical failure rates and mission times are not given, since it still permits the relative comparison of many competing designs. The failure rates are normalized with respect to a reference measure of

complexity and the MTIF is used as the criterion of quality. The most fundamental difference in computer reliability modeling is that between static and dynamic models for the reliability of systems which incorporate protective redundancy. Both classes of models are considered in the following discussion.

The class of static reliability models is suitable for the reliability prediction of systems with static hardware redundancy. The redundant elements are assumed to be permanently connected and to fail statistically independently. They have the same failure rate and are instantaneously available to perform the masking of a failure with unity probability of success. Under these assumptions, the reliability of a redundant system is obtained as the sum of the reliabilities of all distinct configurations that do not lead to system failure. Reliability models of static redundancy are found in handbooks and textbooks of reliability theory and are used for reliability analysis of various redundant structures, e.g., relay contact networks, aircraft frames, etc. [BARL 65]. The principal limitation of the static model in computer reliability modeling is the assumption that the fault-masking action is always successful as long as redundancy is not exhausted. This assumption cannot be justified in systems which employ various forms and combinations of dynamic hardware, software, and time redundancy, and dynamic reliability models have to be created to these systems.

The use of dynamic redundancy requires the success of consecutive fault detection and recovery actions in order to utilize redundant (spare) parts. The use of static reliability models for the dynamic case is equivalent to assuming unity probability of success of both actions. For this reason, very high reliabilities are predicted as the number of spares is increased. Early in the studies of dynamic redundancy it was recognized that imperfect detection and recovery may cause system failure before all spares had been used. The effect of such imperfections was formalized in the dynamic reliability model through the concept of "coverage," defined as the conditional probability

of successful recovery, given that a fault has occurred [BOUR 69]. This model has served as the reference point for subsequent investigations of closed systems, i.e., of those systems in which off-line repair of failed parts is not available, and the system is certain to fail after all redundancy resources have been exhausted.

Recent research has resulted in a general dynamic reliability model which employs Markov modeling techniques and subsumes nearly all models for both static and dynamic redundancy that have been developed to date [NGYW 76, NGYW 77a]. Its principal advantage is that a single efficient computing procedure serves to perform the reliability prediction for any one of a variety of closed systems, including those in which degradation is provided. Extensions to repairable systems and to transient faults also have been made in this model.

A closed fault-tolerant computer system is treated as a set of homogenous closed subsystems, each of which consists of a set of identical modules that are either in active or spare status. "Active" means "participating in the computing process," i.e., a powered spare is not active, although its failure rate is the same as that of the active modules. Since every subsystem must survive in order for the system to survive, the system reliability is the product of the reliability of all subsystems. The modeling effort therefore deals with a closed homogeneous subsystem. The set of modules forming such a subsystem is characterized by the following parameters:

N = Initial number of modules in the active configuration

D = Number of degradations allowed in the active configuration

S = Number of spare modules

Ca = Coverage for recovery from active module failures

Cd = Coverage for recovery from spare module failures

λ = Failure rate of one active module

μ = Failure rate of one spare module
($\mu = \lambda$ if spare is powered)

$\underline{Y}$ = Sequence of allowed degradations of the active configuration

$\underline{CY}$ = Coverage vector for degraded configurations

The parameter $\underline{Y}$ is an integer vector of the form Y = (Y[1], ..., Y[D]), where Y[1], ..., Y[D] are the numbers of active modules remaining in successive degraded active configurations. The coverage vector $\underline{CY}$ has the form CY = (CY[1], ..., CY[D]), in which CY[i] is the coverage associated with the transition to the degraded configuration described by Y[i].

At any given time each module is in one of three possible states: it is in the failed state; it is a good spare (all spares are either powered or unpowered); or it is a member of an active configuration which consists of all those modules currently participating in the computing process. Once a module has failed and the system recovers from the failure (either through static fault-masking or dynamic reconfiguration), it is assumed that the failed module is isolated from the system and will no longer contribute to system reliability or unreliability. This implies that the possibility of compensating failures in voting systems and similar secondary effects are not considered in this model.

In a dynamically redundant subsystem, an active configuration of N modules is supported by a bank of S spare modules. When the spares are exhausted and one more failure of an active module occurs, the subsystem is usually considered as failed. However, in some applications it continues to operate in a degraded mode, i.e., it has a smaller set of active modules (and hence a possible degradation in performance). The abandonment of active modules upon failure continues until the active configuration falls below a specified minimum number of modules, at which time the subsystem fails. The degradation sequence is described by the vector $\underline{Y}$ in the reliability model. Statically redundant subsystems and hybrid-redundant subsystems with a static core also have an active configuration which degrades to some extent before subsystem failure occurs. (For example, a TMR subsystem degrades from 3 to 2 modules upon the first failure). Hence they are treated in the

reliability model in the same manner as the dynamically degrading subsystems.

The condition of a closed subsystem is characterized by a model with a finite number of states, each representing a distinct subsystem configuration which is either good or is failed. For closed subsystems, the goal is to obtain the statistics of the time to first occurrence of subsystem failure. Hence, all failed states are merged into one state denoted by F. A transition out of a good state takes place when a failure occurs in one of the modules. Depending on whether recovery from this failure is successful, the transition will be to another good state or to the failed state. When it is assumed that failure rates are constant and that (with respect to the time scale of reliability prediction) the recovery from a failure is accomplished instantaneously, the model is a Markov model.

The state diagram of Figure 2-2 is the model of the closed fault-tolerant subsystem which is defined by the set of parameters (N, D, S, Ca, Cd, λ, μ, $\underline{Y}$, $\underline{CY}$) explained previously. The subsystem is self-repairing and has provisions for degradation of the active configuration after the spares have been exhausted. The selection of spares occurs in a linear order, and a spare that fails in an unrecoverable mode blocks the use of the spares that follow it in the selection sequence. Furthermore, it destroys the ability to degrade, because the subsystem fails at the time when the unrecoverably failed spare unit is switched into service. This effect is incorporated in the model by transitions to the states with an overbar such as $(\overline{N,S-1})$, $(\overline{N,0})$, etc. The subsystems in the state (N,i) and in the state $(\overline{N,i})$ have the same configuration, but the subsystem at state $(\overline{N,i})$ has lost its ability to degrade because of the existence of a non-recoverable failure in one of the (still unreached) spare modules.

Almost all fault-tolerant system models that have been studied in the past can be represented by this model. Table 2-1 characterizes several of them in the notation described above.

The reliability equation of one closed fault-tolerant subsystem has the form:

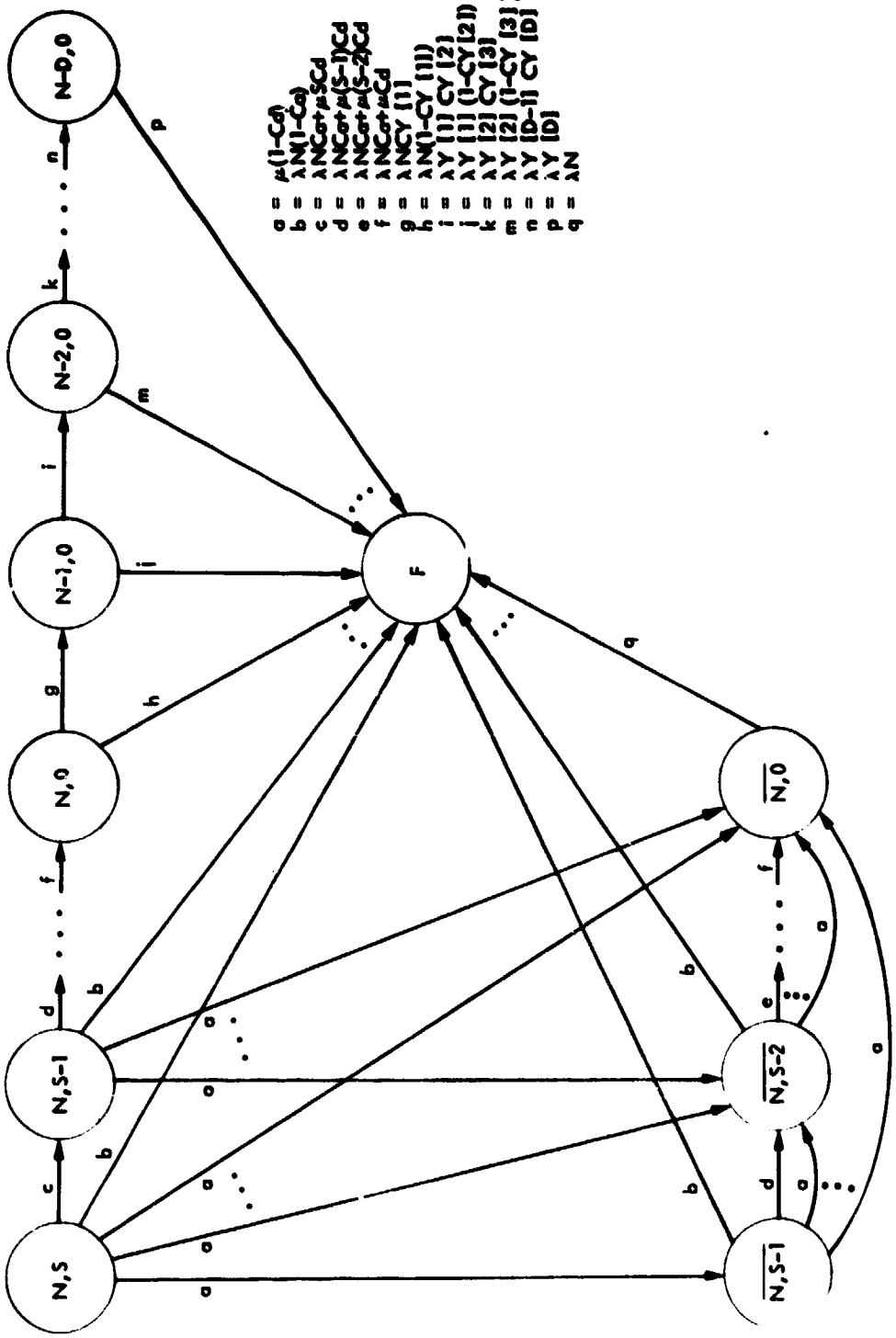


Figure 2-2. Markov Reliability Model for Closed Systems

Table 2-1. Characterization of Several Models of Fault-Tolerant Systems

System	N	D	S	Ca	Cd	λ	μ	Y	Reference
Simplex	n	0	0	1	1	λ	λ		
Static									
TMR	3	1	0	1	1	λ	λ	2	[BOUR 71]
TMR/Simplex	3	1	0	1	1	λ	λ	1	[BOUR 71]
NMR	2n+1	n	0	1	1	λ	λ	2n,...,n+1	[MATH 75a]
NMR/Simplex	2n+1	n	0	1	1	λ	λ	2n-1,...,3,1	[MATH 75b]
Dynamic									
${}_cR_s^q$	q	0	S	C	C	λ	μ		[BOUR 69]
$R^*(N,S,A_c^a,A_c^d)$	N	0	S	A_c^a	A_c^d	λ	μ		[RENN 73a]
K-out-of-N	N	N-K	0	C	C	λ	λ	N-1,...,K	[WYLE 67]
R(2,S)	2	1	S	1	1	λ	μ	1	[RENN 73b]
hybrid									
H(N,S,D)	N	D	S	1	1	λ	μ	N-1,...,N-D	[BRIC 73]
R(N,S)	2n+1	n	S	1	1	λ	μ	2n,...,n+1	[MATH 70]
$R^*_{TMR/Spares}$	3	2	S	1	1	λ	μ	2,1	[TAYL 73]

$$R(t) = X(t) \cdot A \cdot W(t)$$

where:

$$X(t) = (e^{-Y[0]\lambda t}, \ldots, e^{-Y[D]\lambda t}), \text{ with } Y[0] = N$$

$$W(t) = (1, e^{-\mu t}, \ldots, e^{-S\mu t})$$

$$A = \begin{bmatrix} A^0_{S,0} & A^0_{S,S} \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ A^D_{S,0} & A^D_{S,S} \end{bmatrix}$$

The coefficients $A^K_{S,J}$ in the matrix A are functions of the parameters and are computed by an algorithm given in Table 2-2 [NGYW 77a].

Based on the model described above, the UCLA Automated Reliability Interactive Estimation System (ARIES) has been implemented in APL as a set of interactive programs for the modeling of fault-tolerant computers [NGYW 77b]. Generality and efficiency are achieved in ARIES because it is based on the unified solution to the reliability modeling problem. To achieve flexibility, the user is provided not only with functions for evaluating the reliability measures of interest, but also with programs to create, modify and examine representations of the systems which are being designed.

The Markov model for closed systems shows that their reliability equations have the standard form:

$$R(t) = \sum_i A_i e^{-\sigma_i t}$$

where the σ_i are simple functions of the modeling parameters and the A_i can be efficiently computed. By applying Markov modeling techniques to repairable systems, the same standard form for their reliability equations is obtained, but now the σ_i must be computed as eigenvalues of the transition probability matrix of the Markov model and the A_i need a more general and less efficient procedure. The reliability analysis of both

Table 2-2. Algorithm for the Components of Matrix A

Step (1) Start with $A_0^0 = 1$. Go to Step (2) if $D = 0$.

For $I = 1$ to D, iterate the following computations:

$$A_J^I = \frac{CY[D - I + 1] \cdot y[D \quad I] \cdot A_J^{I-1}}{Y[D - I] - Y[D - J]} \qquad \text{for } J = 0, \ldots, I-1$$

$$A_I^I = 1 - \sum_{J=0}^{I-1} A_J^I$$

Step (2) Set $A_{0,0} = 1$

Using results of Step (1), set $A_{0,0}^K = A_{D-K}^D$, for $K = 0, \ldots, D$.

For $M = 1$ to S, iterate the following computations:

Step (2a)

$$A_{M,J} = \frac{(NCa\lambda + MCd\mu)\, A_{M-1,J} + (1 - Cd) \sum_{I=J}^{M-1} A_{I,J}}{(M - J)\mu}$$

for $0 \leq J < M$

$$A_{M,M} = 1 - \sum_{J=0}^{M-1} A_{M,J}$$

Step (2b) $0 < K \leq D$: $A_{M,J}^K = \dfrac{(NCa\lambda + MCd\mu)A_{M-1,J}^K}{(N - Y[K])\lambda + (M - J)\mu}$ for $0 \leq J < M$

$$A_{M,M}^K = 0$$

Step (2c)

$K = 0$: $A_{M,J}^0 = \dfrac{(NCa\lambda + MCD\mu)A_{M-1,J} + (1 - Cd)\mu \sum_{I=J}^{M-1} A_{I,J}}{(M - J)\mu}$

for $0 \leq J < M$

$$A_{M,M}^0 = 1 - \sum_{(K,J) \neq (0,M)} A_{M,J}^K$$

closed and repairable systems has many common properties that have allowed an extension of ARIES to include repairable fault-tolerant systems as well.

The repairable systems modeled by ARIES are the closed systems in the Markov reliability model which have been made repairable by the presence of one or more repairmen [NGYW 77a]. Hence, they are modeled by the same set of parameters (N, D, S, Ca, Cd, λ, μ, $\underline{Y}$, $\underline{CY}$) as closed systems, plus two more parameters (M, Ψ), where M is the number of repairmen and Ψ is the repair rate of each repairman.

2.4.2 Analytic Modeling: Transient Faults

The next step to be taken in modeling is to address the problem of transient faults [NGYW 76]. These cause system failures by damaging the information content of the system during their presence. This damage will be permanent and will eventually lead to irrecoverable errors in the system unless some means of recovery is provided. Recovery in this case consists of a restoration of the information structure so that the system can continue to function properly. The hardware remains intact and the full capability of the machine is retained, in contrast to permanent fault recovery where the system degrades in performance unless spares are used to replace faulty modules.

The methods to effect recovery from suspected transient faults usually consist of a number of successively more difficult recovery phases. For example, a system may use the sequence of an initial delay, instruction retry, program rollback, and system restart as a four-phase recovery procedure. The next phase is entered if the current phase fails to accomplish a satisfactory recovery.

The processes which generate transient faults are difficult to characterize. The model adopts the viewpoint that the transient fault environment can be characterized by two fundamental parameters--transient arrival rate and transient duration [AVIZ 75a]. It is assumed that transient arrival is a Poisson process with a constant arrival rate and that each transient fault has a duration which is independently distributed according to an exponential law. These

assumptions appear to be consistent with the limited number of observations on transients available in current literature and have the advantage of being more readily mathematically tractable than other possible choices. The two parameters modeling the transient fault environment under these assumptions are defined as follows:

τ = transient fault arrival rate for one module

$\bar{D}$ = mean duration of each transient

A transient recovery process may fail because of several reasons. The model deals with four causes of failure: one is excessive duration, which is a function of τ and $\bar{D}$, while the other three are characterized by the parameters recoverability r, effectiveness E, and interference rate ρ. All four are discussed below.

The first cause is occurrence of persistent transients. They are transients that last throughout an entire phase of a recovery action, causing that phase of the recovery action to fail. A very long transient will lead to unsuccessful outcome of the entire transient recovery effort. Then the transient fault will be treated as permanent by the system and permanent fault recovery actions will be initiated. The probability of a persistent transient depends on the arrival rate τ and mean duration $\bar{D}$ of transient faults.

The second cause is a catastrophic fault. Such a fault occurs when the transient fault damages insufficiently protected critical information. Also, faults that are not detected soon enough after their occurrence can lead to so much information damage ("memory mutilation") as to make recovery impossible. Furthermore, real-time systems have certain tasks which must be accomplished within strict time limits. Delay of these tasks by a transient fault also may lead to a system crash. The probability of these and other possible catastrophic faults is modeled by the recoverability parameter r which is defined as the conditional probability:

$$r \quad \text{Prob (fault is not catastrophic} \mid \text{fault occurs).}$$

Since the effects of both permanent and transient faults are similar in most systems and are about as likely to cause catastrophic failures, one value of r is used to model both types of catastrophic faults.

The third cause of unsuccessful recovery for a given recovery phase is the ineffectiveness of the recovery technique employed. For example, it has been estimated that instruction retry as a transient fault recovery technique is effective only in approximately half of all cases [CART 64]. The effectiveness of a particular recovery phase is modeled by the effectiveness parameter E which is defined as the conditional probability:

$$E \equiv \text{Prob (recovery action is successful} \mid \text{it is initiated against a noncatastrophic transient fault).}$$

The fourth cause is interference which occurs when a second independent fault (transient or permanent) interrupts the function being performed to effect a recovery. How the system behaves in the presence of such interference depends on the recovery capability that is built into a system. A conservative assumption is that interference, like catastrophic failures, will always lead to a system crash. The probability of interference depends on the duration of recovery, and on the complexity of the recovery elements that must remain fault-free in order to carry out the recovery action. The latter is modeled by the interference rate, defined as:

$\rho \equiv$ failure rate of the recovery element hardware. This hardware includes both dedicated recovery hardware elements and those elements that are used to store, deliver, and execute recovery software.

Given the preceding parameters, transient fault recovery can be modeled as a part of the general model; that is, it is also modeled on a subsystem basis, since each subsystem may have different recovery requirements and a separate recovery strategy may apply for each. The recovery strategy is a multiphase recovery process which executes n successive recovery phases, as shown in Figure 2-3. Transition to the next phase takes place if the present phase is not effective. The recovery process is completed and normal processing resumes if a successful recovery is achieved during the present phase. The system can crash during the present phase due to interference. (A "crash" is a failure of the programs to continue correct execution.) If neither a crash nor a recovery occurs in all n phases, then the transient recovery

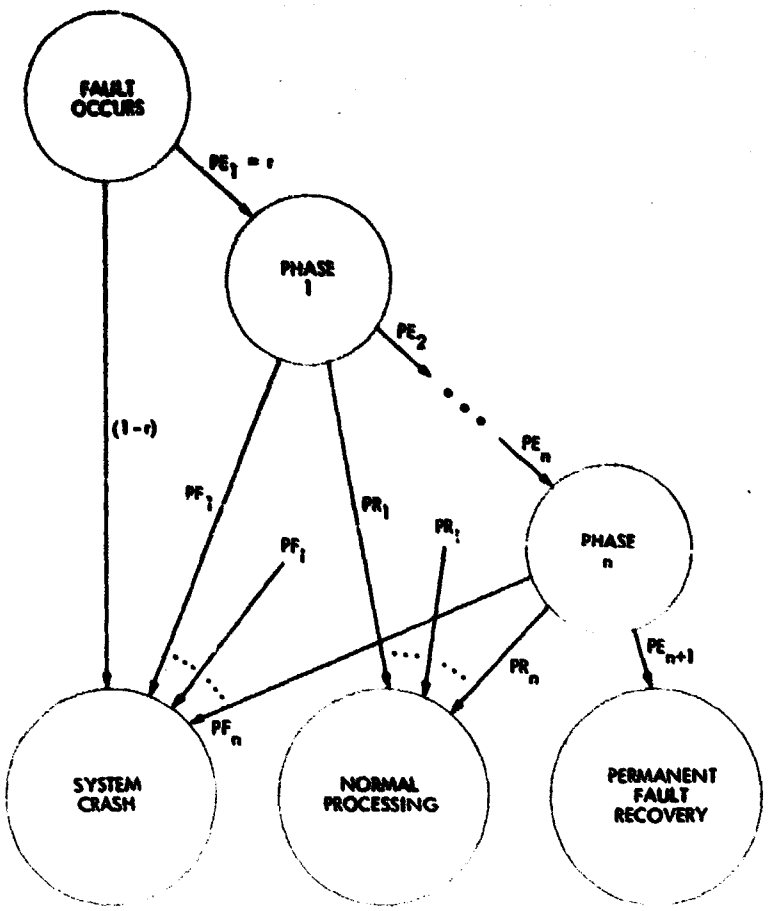


Figure 2-3. Transient Fault Recovery Process

process is considered to have been unsuccessful (because the fault pe" ists) and permanent fault recovery is initiated.

The model employs (for i = 1, ..., n), the following conditional probabilities:

PE_i = Prcb (system enters i-th recovery phase | fault occurs)

PR_i = Prob (system recovers in i-th recovery phase | fault occurs)

PF_i = Prob (system crashes in i-th recovery phase | fault occurs).

The sequence of events in a transient fault recovery process is depicted in Figure 2-3, which shows its three outcomes. They are parameterized

by the following three conditional probabilities, which apply to the transient recovery process:

$$C_T \ (\equiv \text{Transient Coverage}) = \sum_{i=1}^{n} PR_i$$

= Prob (Transient recovery succeeds | fault occurs)

$$L_T \ (\equiv \text{Leakage}) = PE_{n+1}$$

= Prob (Fault is treated as permanent | fault occurs)

$$F_T \ (\equiv \text{Probability of a crash}) = (1 - r) + \sum_{i=1}^{n} PF_i$$

= Prob (System crashes during recovery | fault occurs).

Because a system usually cannot immediately distinguish whether a detected fault is transient or permanent, it is assumed that the transient fault recovery is the first process initiated. This assumption is reflected in the definition of the above parameters by making them conditional on the occurrence of any fault, transient or permanent. The parameters C_T, L_T, and F_T give the relative probabilities of the three possible outcomes of the transient recovery process and thus determine the reliability of the system in the presence of transient faults. To complete the choice of modeling parameters, it is necessary to define:

E_i = Effectiveness of the i-th recovery action

T_i = Duration of the i-th recovery action

In the general case, T_i is a random variable. In order to limit the complexity of the model, the assumption is made that it is a constant, which would be an upper bound. It is also postulated that the first stage of any recovery strategy is an intentional delay of duration T_D, in order to allow the transient fault to subside. Then $T_1 = T_D$ and $E_1 = 0$ since there will be no active recovery action during the delay (recovery phase 1). The transient reliability measures C_T, L_T, and F_T are computed from the basic parameters of the subsystem by the use of some simple probability relations, as shown in Table 2-3.

Table 2-3. Derivation of Transient Reliability Measures

- PF_i = Prob (phase i is entered) × Prob (interference in phase i)

$$PF_i = PE_i \times (1 - e^{-\rho T_i})$$

- PR_i = Prob (phase i is entered) × Prob (fault is a transient)
  × Prob (recovery action is effective) × Prob (no interference)
  × Prob (no recurrence of fault in phase i)
  × Prob (duration of transient does not extend into phase i)

$$PR_i = PE_i \times K_i \times E_i \times e^{-\rho T_i} \times e^{-(\tau+\lambda)T} \times \left[1 - e^{-\frac{1}{D}(T_1 + \ldots + T_{i-1})}\right]$$

- $PE_{i+1} = PE_i - PF_i - PR_i$; $PE_1 = r$

The factor K_i = Prob (fault is a transient | phase i) is a probability conditional on entry to phase i of the recovery process and decreases as i increases. The reason is that with increasing knowledge that the fault has not been eliminated by the preceding recovery mechanisms, there is more likelihood that it is a permanent instead of a transient. To estimate K_i, define

A_i = Prob (recovery phase i is entered | fault is a permanent)

B_i = Prob (recovery phase i is entered | fault is a transient)

We assume $A_1 = B_1 = r$, that is, a catastrophic fault will not cause entry to phase 1, but will enter the system failure state immediately. The following relations also hold:

$$A_{i+1} = A_i \times \text{Prob (System does not crash in phase } i) = A_i e^{-\rho T_i}$$

$$B_{i+1} = B_i \times \text{Prob (System does not crash, but there is no recovery in phase } i)$$

$$= B_i \times e^{-\rho T_i} \times \left\{1 - E_i e^{-(\tau+\lambda)T_i}\left[1 - e^{-\frac{1}{D}(T_1 + \ldots + T_{i-1})}\right]\right\}$$

Then

$$K_i = \frac{\tau B_i}{\lambda A_i + \tau B_i}$$

where λ, τ are respectively the permanent and transient failure rates of one subsystem module.

The reliability model of Figure 2-2 does not include transient fault recovery. This limitation is removed by integrating the transient fault recovery model into the unified model [NGYW 76]. Figure 2-4 shows, on a localized basis, the incorporation of the transient fault recovery model into the reliability model of Figure 2-2. Two additional states are introduced between each pair of successive operational states of the subsystem to represent the existence of the transient and permanent fault recovery processes. In addition to the original set of parameters, transitions between states are also governed by the three transient fault recovery parameters: C_T, L_T and F_T. It is assumed that transients have no effect on the status of spare modules; hence the transitions between states that are caused by spare module failures remain the same. Although the system spends a finite amount of time in these two recovery states, for all practical purposes it can be assumed that the recovery process is instantaneous, because even in the worst case the recovery time is several orders of magnitude smaller than the average time between faults in the hardware. With this assumption, the two recovery states are merged into the operational states and Figure 2-4 becomes Figure 2-5. The general model of Figure 2-2 is preserved when the transient fault recovery model is incorporated. The main effect of this incorporation is to change the effective failure rate of each module from λ to λ' and the effective coverage factor from Ca to Ca' as given in Figure 2-5. The derivation of λ' and Ca' follows from Figure 2-4.

Because the general model of Figure 2-2 is preserved, the same efficient computational procedure can be applied in those cases where transient fault modeling is desired, with the obvious modification that λ and Ca must now be replaced by λ' and Ca'. The programming system ARIES has been extended to model transient fault recovery. Based on a characterization of transient fault recovery in a subsystem by means of the parameters τ, $\bar{D}$, ρ, r, E_i, T_i and T_D. ARIES estimates the transient fault recovery parameters C_T, L_T and F_T from which an efficient reliability estimation of a subsystem is mixed transient and permanent fault environments can be made [NGYW 77b].

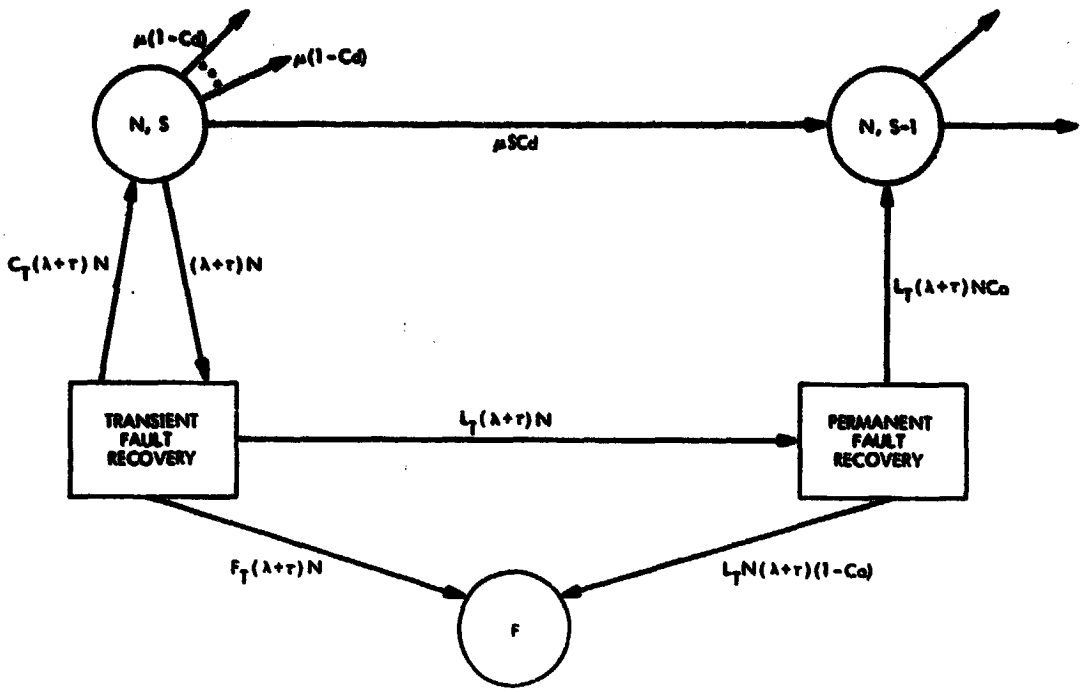


Figure 2-4. Transient Recovery in the Markov Model

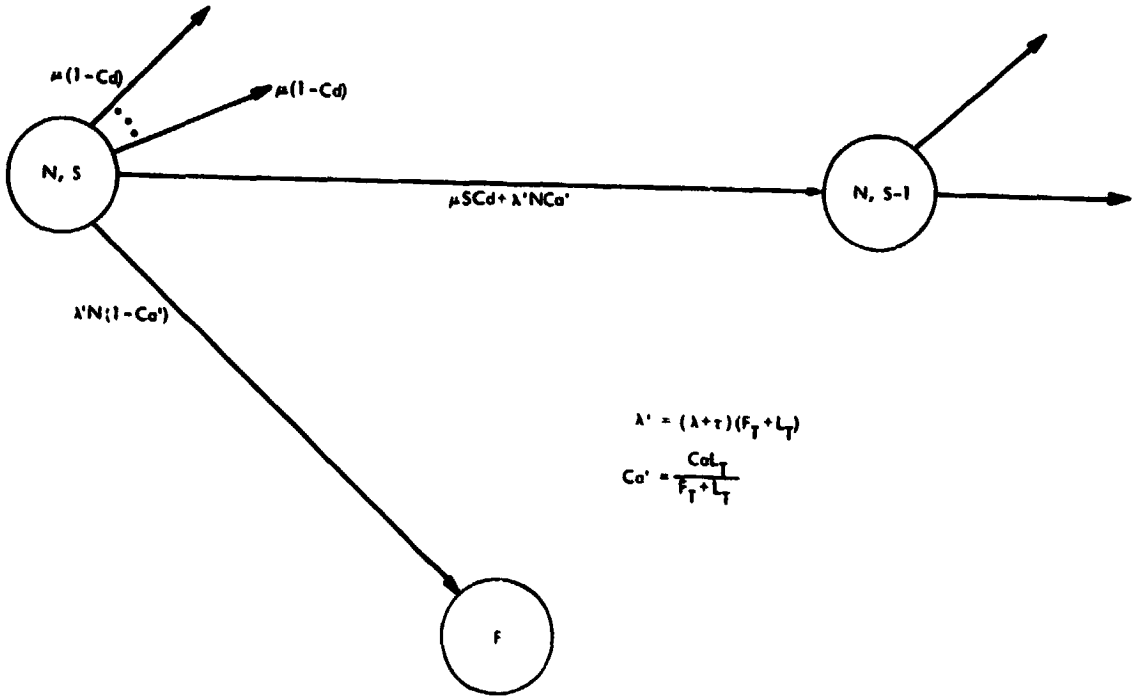


Figure 2-5. Equivalent Form of the Markov Model

2.4.3 Heuristic Approaches: Simulation and Experiments

Simulation and experimentation with a hardware prototype are two approaches to heuristic prediction of reliability. Although their use is more costly and time-consuming than that of analytic models, these methods are essential when the analytic models do not adequately represent the complex structure of the system or the nature of the expected faults. Furthermore, the users of systems in various failure-critical applications often insist on heuristic validation of the initial analytic results prior to the production and use of a system.

An accurate description of the system and detailed characterization of faults are the principal prerequisites when simulation is employed to derive the reliability estimates for the computer. Modern simulation programs include provisions to model both permanent and transient faults, and to consider the hardware-software interaction by representing a variety of recovery algorithms [LEVY 75]. An important early use of simulation was the reliability prediction of TMR logic in the SATURN V guidance computer [ANDE 67].

Experimental reliability prediction using a hardware prototype requires a large investment of effort in constructing the prototype, but avoids the inaccuracies which may occur in postulating the fault effects in a simulated model of the system. An example is the experimental fault-tolerant JPL-STAR computer. In this computer an electronic "black box" was used to inject faults of adjustable duration and extent at selected points in the hardware of the system during its operation [AVIZ 72]. Statistical data on the cases in which recovery did not succeed was automatically collected and processed. The data was also used to derive estimates of the coverage parameters for analytic modeling. Several weaknesses in the fault-tolerance implementation of the original design were identified and eliminated during the experiments. The stability of recovery algorithms was studied under multiple-fault and repeated-fault conditions, and the performance of system software was extensively tested.

The current rapid advances in the design of novel and complex fault-tolerant systems have overtaken the capabilities of

analytic modeling. As a consequence, experimental reliability prediction remains a very important area for further development and application.

2.5 TOLERANCE OF MAN-MADE FAULTS

Man-made faults are all non-physical faults that occur because of human mistakes, i.e., execution of improper actions or absence of expected actions during the procedures of specification, design, detailed implementation (construction or programming), modification, maintenance, and use of information processing systems. They do not include physical faults that are consequences of human actions. The manifestations of such physical faults are the same as those caused by natural phenomena; for this reason they are treated by the same techniques of fault-tolerance and belong in the same category as all other physical faults. Man-made faults include the non-physical faults caused by imperfections in various design, programming, and maintenance tools, such as compilers, assemblers, design automation programs, maintenance and operation manuals, testing procedures and devices, etc.

For the purpose of systematic discussion, it is convenient to partition man-made faults into the classes of design faults and interaction faults. Design faults are the faults that are introduced into the system during various phases of implementation: Specification, design, programming, translation to machine code, detailed logic design and layout of logic circuits, interconnection of hardware elements, and later modifications of hardware and software. The causes of design faults are twofold: incomplete, ambiguous, or erroneous specifications, and mistakes committed during the various phases of translation of a specification into the final implementations, i.e., assemblies of interconnected hardware elements and arrays of digitally represented symbols.

Interaction faults are faults that are introduced into the system via man/machine interfaces during operation or maintenance phases by operator action that is not appropriate to the current state of the system. They are caused typically either by a misunderstanding of the operator's manuals or by typographical errors that occur while information is entered into the system.

The problem of man-made faults has remained of consistently great concern to the designers and users of information processing systems from the specification of the first system to the present. Some complex and costly systems have never reached an operating condition because design faults could not be eliminated or controlled (tolerated) within the existing time limits and cost constraints. Many other systems have experienced severe delays in delivery and major cost overruns. In a few cases the question of the possible existence of latent design faults has spilled over from technological and economic considerations into politics and public controversy. A very prominent illustration of such an event is the recent controversy in the U.S. regarding the possibility of unreliable behavior of the ABM (anti-ballistic missile) defense computer system.

In contrast to physical faults, the problems of man-made faults have not been suddenly alleviated by a major technological breakthrough similar to the invention of semiconductor and magnetic core components. Advances in the understanding and ability to handle man-made faults have come at a slow and steady rate, and they have barely kept pace with the rapidly growing complexity of systems and the increasing demands for near perfectly fault-free system behavior in numerous critical applications, in some of which human lives are endangered by fault-induced system failures.

2.5.1 Design Faults

An overview of the approaches used to handle faults from the origins of machine computing to the present shows that a priori fault elimination (fault-avoidance) has been the dominant choice for the handling of design faults that are introduced during specification, design, construction, programming, and modification of both hardware and software [ICRS 75], [NELS 75]. An all-out effort to eliminate design faults takes place before the system is first put into regular service or returned to use after a modification.

The approaches taken to assure design fault elimination have originated both in theoretical studies and in problem-solving approaches developed from direct experience. The main theoretical developments in

this area are proof-of-correctness techniques [LOND 75] and mathematical models for software reliability prediction [SHOO 73], [MORA 75]. The practice-originated "software engineering" techniques include procedures for the collection and analysis of fault data, management procedures for software development, tools and techniques for software design, such as specification languages and the structured programming approach, software verification and validation techniques [NELS 75], and digital-logic simulation techniques for hardware design verification [SZYG 76], [BUTL 74].

Despite all of the above techniques for fault elimination, left-over design faults have been observed in most systems during operation. For this reason most systems have been provided with emergency procedures to detect error states that may be due to design faults, to record them, and to bring the system to a state in which external assistance may be brought in to complete the analysis of the condition and to reinitiate operation. While these emergency procedures are not unlike some fault-tolerance techniques for physical faults, the function that is accomplished is only the "shutdown" function with respect to either a part of the system or the entire system.

More complete fault-tolerance of design faults has not yet been introduced into existing computer systems, and only very recently have some research efforts been started to explore this problem in depth. Because of the existence of much more extensive research and practical experience with the tolerance of physical faults, it is interesting to look for transferability of concepts and techniques. The principal difference between physical and design faults is that physical faults in hardware occur after the start of the computing process, while design faults in software (and hardware, as well) are present at the start, but become disruptive only at a later time. However, modifications or corrections of discovered design faults occasionally lead to new design faults, and therefore the discoveries of software and hardware design faults may be expected throughout the useful life of any large system, similar to the occurrence of physical faults. This practically verified observation establishes a relationship between the methodologies for dealing with physical faults and design faults: the methods of protective redundancy that have proven successful in the

tolerance of physical faults may be transferable to provide tolerance of design faults as well. Three aspects of relevance of physical fault-tolerance can be identified [AVIZ 75b]:

(1) The contribution of physical fault-tolerance techniques in identifying and isolating design faults.

(2) The common aspects of fault-tolerance that are equally relevant to physical and design faults.

(3) The transfer of physical fault-tolerance techniques and experience of software design faults, considering:

  (a) the applicability of software,

  (b) the potential advantages of software fault-tolerance,

  (c) the cost of its use, compared against the traditional fault-avoidance techniques.

First, the presence of physical fault-tolerance is directly useful in handling design faults because it provides the means to identify those cases of abnormal system behavior that are due to physical faults. Furthermore, extensions of physical fault-tolerance techniques may be applicable to provide hardware-controlled protection of software and the data base against attempts to interfere with its operation and to access privileged information.

Second, an area in which a common ground exists for physical and design fault-tolerance efforts is the analytic modeling and quantitative prediction of system reliability. Recent work on software reliability models [SHOO 73], [MORA 75] indicates the possibility of mutual reinforcement that would lead to the development of analytical models for the total system reliability, including both the physical fault and design fault aspects.

Third, the redundancy techniques that have been successful in handling physical faults may be transferable to design fault-tolerance. Both the static and the dynamic hardware redundancy approaches have their counterparts in software fault-tolerance. In the static case (called N-version programming), two or more programs are generated independently

and then are operated concurrently on multiple copies of the fault-tolerant hardware [AVIZ 77b]. Comparison or majority voting at specified points is employed to detect or correct the effects of design faults. Systems such as SIFT [WENS 76], the Symmetric Multiprocessor [HOPK 75], and the Space Shuttle Computer System [SKLA 76], are especially suitable for such N-version programming. The dynamic case uses the equivalent of standby sparing, in which acceptance tests serve to detect design faults and to initiate a switchover to an alternate software module [RAND 75], [HECH 76]. An extension of the above techniques to hardware design fault-tolerance is also feasible: functionally identical copies of modules then must be independently designed and manufactured by separate organizations in order to avoid the occurrence of identical design faults in all copies.

The state of the art in fault-tolerance of design faults resembles that of physical fault-tolerance in the early 1960's. The cost and the effectiveness of the design fault-tolerance approaches remain to be investigated, and the techniques require much further development and experimentation. The success of fault-tolerance of physical faults, however, does indicate very strongly that design fault-tolerance cannot be safely ignored solely because of the past tradition of fault-avoidance in this field.

2.5.2 Interaction Faults

The possibility of introducing man-made faults also exists via man/machine interaction during system operation. The control of such interaction faults has been implemented primarily by means of operator training and by providing complete guidelines in operation and maintenance manuals. This approach corresponds to the fault-avoidance approach for physical and design faults. The demands on the operator have been reduced by the development of increasingly more sophisticated operating systems. However, interaction faults have remained a major problem area in system operation.

Fault-tolerance approaches to interaction faults have remained confined to immediate practical solutions to observed problems. The principal goal here is the implementation of the detection function,

which allows the system to reject apparently incorrect operator inputs. The main methods are consistency checks, requirements for appropriate passwords, and coded data entry. In some very critical cases, two or more operators are employed whose input commands and data must agree in order to be accepted by the system.

2.6 CURRENT PROBLEMS AND PROSPECTS FOR THE FUTURE

2.6.1 Reasons for Fault-Tolerance

At the present time, we can identify several reasons for the acceptance and general use of full fault-tolerance (without manual intervention) in information processing system of the future. The main reasons are:

(1) The need to minimize the risks associated with computer failures in systems in which the failures either endanger human lives, or threaten to cause heavy economic losses to the users. Examples of the first class are systems for patient monitoring in hospitals, for air traffic control, and for guidance and control of high-speed vehicles. In the second class are systems to control power generation and distribution, to control processes in automated factories, to handle financial transactions, etc.

(2) The need for reliable computing in environments that do not allow access for manual maintenance, such as space and undersea locations, and other locations in which access is either impossible or excessively costly.

(3) The need for almost uninterrupted operation of real-time systems in which manual intervention creates unacceptable delays.

(4) The possibility of lower initial cost (for a given reliability goal) than a system that depends on fault-avoidance. This may occur in those cases in which

fault-tolerance allows the use of less costly components, or reduces the cost of design-fault elimination prior to system delivery.

(5) The possibility of a lower life-cycle cost than a system with manual maintenance requirements. Fault-tolerance can reduce maintenance to a scheduled off-line replacement of disconnected elements (or an exchange by mail!), and eliminate the costs associated with the unavailability of a system between failure and completion of repair.

(6) The psychological support to system users provided by the knowledge that fault-tolerance is incorporated into the system on which they depend for their safety or economic benefit.

2.6.2 A Design Methodology

Research results and design experience lead us to suggest that the introduction of fault-tolerance can be accomplished by following a systematic procedure:

(1) Performance requirements are established and system architecture is specified with the initial assumption that faults will not occur.

(2) Classes of faults that are to be tolerated in the design are identified, and the extent of tolerance is specified for each class of faults.

(3) Cost-effective methods of protective redundancy (time, hardware, software) are chosen to cover every class of faults identified above, and system architecture is modified to incorporate the redundancy.

(4) Analytic or experimental reliability prediction techniques are employed to evaluate the fault-tolerance that is provided by redundancy.

(5) Checkout methods are devised to test all redundancy features. Where applicable, fault-tolerance is extended to effect automatic maintenance of peripheral systems that are connected to or controlled by the computer.

Design experience has shown that several iterations of (3) and (4) may be necessary to arrive at a satisfactory fault-tolerant system architecture.

2.6.3 Current Roadblocks

In view of the potential benefits of full fault-tolerance, it is inevitable to ask: "Why is there so relatively little fault-tolerance in the computer systems of the present generation?" The obstacles to the appearance of full fault-tolerance are rather diverse. Some of the more obvious problem areas are identified below.

(1) Lack of Continuity. Some fault-tolerance techniques developed for first-generation computers (for physical faults) were discarded in the second generation because of much higher reliability of semiconductor and magnetic-core components. Later, many ad hoc solutions were not openly documented because of their trade secret status, leading to the re-invention of good solutions as well as the repetition of many mistakes of the past.

(2) Lack of Cost/Benefit Measures. Thus far, there are no general methods for a convenient quantitative assessment of the benefits (in terms of life-cycle cost reduction) of fault-tolerance. The initial extra cost which is due to the various redundancy techniques is much more directly evident and tends to bias a large class of users (who do not have an absolute requirement) in favor of systems without fault-tolerance.

(3) Lack of Specifications and Acceptance Tests. The user community at large still does not have a sufficient

knowledge of the properties and limitations of fault-tolerance. As a consequence, specifications of reliability are insufficiently precise and virtually unverifiable in advance of system use. For example, most reliability requirements for a given time interval do not specify the classes of faults and do not state what constitutes acceptable recovery. For another example, MTBF specifications do not explicitly deal with fault classes (e.g., transients, design faults) and recovery requirements, and also ignore the differences between redundant and nonredundant designs. Extremely high reliability and MTBF predictions are sometimes offered without stating the implicit assumptions of a static reliability model and a very limited class of faults. For contrast, consider speed requirements in instructions/second, which can be stated and tested for acceptance quite precisely.

(4) Fragmentation of Efforts. Efforts to increase reliability of computing originate within several disciplines of theory and practical computer engineering. These include computer system architecture, software engineering, testing and design verification, design of data base management systems, computer networks and communication systems, component and packaging engineering, field operation and maintenance, and others. Although they all have a common end goal, the efforts have remained largely disjoint. A definite lack of a common viewpoint and of systematic communciation is evident at the present time. There is also a real gap between the results of theoretical investigations and practical engineering solutions to fault-tolerance problems.

(5) Inertia in the Design Process. Introduction of fault-tolerance requires an early committment and a significant departure from traditional evolutionary design of computer product lines, in which compatibility of

software is usually a dominant factor. While the number of fault-tolerance techniques to serve as maintenance aids has been increasing, none of the major manufacturers has yet announced a fully fault-tolerant line of computers. The only fault-tolerant systems that were actually delivered were custom-made products for special requirements.

(6) Resistance to Potential Impact. Successful introduction of fault-tolerance may cause some de-emphasis of several currently flourishing activities. Examples are the production of ultra-reliable components, the business of providing manual maintenance and the activities associated with the *a priori* verification of software. It is not unexpected to encounter skepticism about fault-tolerance from the advocates and suppliers of those techniques.

In conclusion, we note that while most of the above-enumerated difficulties are common to many disciplines of computer engineering and computer science, they reach probably their greatest severity in the studies and implementation of fault-tolerance.

2.6.4 Goals and Prospects

The preceding list of problem areas also serves as a guide for the selection of goals for research, development and implementation of systems. Major goals in fault-tolerance for the immediate future are:

(1) The development and acceptance among designers, analysts, and users of information processing systems of an integrated viewpoint of fault-tolerance as an attainable and necessary attribute of a good system.

(2) The development of precise quantitative methods for the specification, acceptance testing, and cost/benefit analysis of fault-tolerant systems.

(3) The design, construction, and testing of experimental fault-tolerant systems. Such systems are absolutely essential, since they serve as vehicles for the validation of new ideas, for the development and refinement of performance specifications and acceptance tests, and for the education of potential users, proving that such systems can be practically delivered.

(4) Continuing investigations of the new frontiers in fault-tolerance techniques, especially the tolerance of design faults in software and hardware, modeling and analysis of complete systems, advanced degradation techniques for large systems, and fault-tolerance for interaction faults. Another stimulating new idea is the possible use of artificial intelligence techniques to implement fault-tolerance [GOLD 75].

The preceding discussion has shown that fault-tolerant computing is still a young, largely unexplored and undeveloped discipline. The accelerating progress in both theory and implementation indicates that the ability to tolerate a large class of physical, design, and interaction faults will be taken for granted in the computer systems of the 1990's, just as the ability to execute a large class of programs is taken for granted in the computer systems of today.

SECTION 3

OBJECTIVES AND ARCHITECTURE SELECTION

The purpose of this section is to describe the assumptions and tradeoffs which led to the selected building block-SCCM architecture. Key objectives of the study are:

(1) to examine and evaluate architectural techniques by which fault-tolerance can be incorporated in next-generation computer systems;

(2) to determine requirements for VLSI circuitry which will be required; and,

(3) to investigate the feasibility of incorporating fault-tolerance as an integral part of future USN building-block computer programs.

The complexity of modern military systems has led to a significant problem of maintenance. Equipment failures lead to a reduction in operational readiness, and maintenance support is a major element in the life-cycle costs of a number of weapons systems. This study is directed toward the routine use of automated redundancy techniques to greatly reduce and simplify system maintenance requirements.

The starting point to achieve this goal is the core electronics portion of complex systems. A technology of fault-tolerant computing has been developed which provides correct computer operation in the presence of internal faults by the use of redundancy and automated repair. Using these techniques, computers can be developed at relatively low cost which provide long-term reliability and which can be utilized to automate system diagnosis and repair by:

(1) diagnosing faults and specifying modular replacement in external subsystems, or

(2) performing automated system repairs to achieve maintenance-free missions.

The scope of this work unit is limited to the digital computing system and those fault-tolerance techniques which can be utilized in the context of a computer building-block development program using next generation VLSI technology.

Although the theoretical groundwork for fault-tolerant computing has been rather well developed, the use of such machines has been limited to a very small number of special applications. The Apollo guidance computer, OAO spacecraft, and ESS telephone switching systems are the primary examples which are most often quoted. These are all custom machines for a specific application.

This study is directed at the question: "What is required to enable the routine use of fault-tolerant computing in a wide range of applications?" First, the requirement for fault-free computing must exist, e.g., the system designer must express a need for correct answers and no unscheduled downtime. But in order to levy this requirement, the designer must be assured of two things:

(1) that the cost of a fault-tolerant design is lower than the cost of an occasional computer failure.

(2) that the risk is acceptable, i.e., that the fault-tolerant computer will be delivered in time and work as specified.

In order to achieve the twin goals of low cost and risk it is best to avoid custom designed computers, and concentrate on machines which are already in wide usage. Not only is extensive software available, but existing chip sets such as the TI 9900, LSI 11, and the 8086 have been characterized and tested through widespread use.

Thus, we have concentrated on the use of existing machines in fault-tolerant configurations. In order to satisfy the project user with regard to risk, the resulting architecture should be straight forward and operate in a fashion that can be readily understood. It should be compatible, as much as possible, with existing standardized components, interconnections, and busing formats. And, indeed, the fault-tolerant

architecture should be capable of a wide range of applications so that it can be included in a future standards program. Risk, and often cost, is lowest when a project can use components and architectures which have previous operational experience.

In order to achieve acceptable costs, the surrounding circuits (used to combine processors and memories into a fault-tolerant configuration) must be reduced to a small number of standard elements and implemented in VLSI packages. At the current state of the art, a microcomputer may require 50 LSI chips, while the surrounding circuitry for fault-tolerance and interconnects may require several hundred MSI circuits. In order to make fault-tolerance attractive to the user, those surrounding circuits must be packaged as a few standard VLSI components.

The primary objective of this study is to develop and verify a small set of building block VLSI circuits which can be used to combine existing processors and memories into fault-tolerant computer configurations.

3.1 REQUIREMENTS FOR FAULT-TOLERANT BUILDING-BLOCK COMPUTERS (FTBBC)

Fault-tolerance requirements are derived from a set of assumptions on the applications in which the FTBBC will be used. These assumptions on applications and the resulting requirements are listed below:

(1) The fault-tolerant computer(s) will be used in a wide range of applications and, in some cases, will perform vital functions (such as system-level redundancy management).

   (a) Thus, over a user-prescribed maintenance interval the reliability should be quite high--99% or greater.

   (b) Wide variations in the maintenance interval should be readily accommodated by adding or deleting redundant elements.

(2) The system containing the computer(s) will have an operational life of a number of years.

(a) The fault-detection and recovery mechanisms of the FTBBC must be thorough and nearly perfect to attain reliability over a long period of time. This is independent of how short a maintenance interval is chosen or how many spares are employed. Reliability modeling studies have shown that "coverage" (the probability of a correct recovery, given that a fault occurs) must approach unity to achieve long-life without computational errors or down time.

(3) It is assumed that for most systems, regularly scheduled maintenance is possible. The computer will "fix itself" by replacing faulty modules with spares; and the discarded faulty modules will be replaced by a repairman at the scheduled maintenance time. In this mode of operation, the scheduled maintenance is best described as preventive maintenance since the computer is still running. It is important, however, that the scheduled maintenance costs be minimized. Therefore:

(a) Redundancy should be applied in an efficient fashion to minimize the number of parts which can fail, and to reduce initial procurement costs.

(b) The fault-tolerant computer(s) should be capable of diagnosing its own faults to a level which facilitates off-line repair.

(4) For applications where human repair is not possible, the maintenance interval will be specified to be the total operational life of the computer(s) and an appropriate number of spare elements shall be employed to achieve the desired reliability.

(5) **The functions to be performed by the computer(s) will be vital to the proper operation of its host system.**

(a) **The computer(s) should not generate erroneous outputs between occurrence and correction of a fault. This implies concurrent fault detection in all parts of the computer(s).**

(6) **Systems have a wide range of requirements on the allowable time-outage while the computer(s) is recovering from a fault.**

(a) **Capability must be provided to allow for a recovery time in milliseconds which is assumed to be a worst-case requirement.**

In short, the FTBBC architecture must have concurrent fault detection to attain high coverage and a rapid recovery time. The structure must also be modularized to allow an arbitrary number of spare elements and simplify replacement procedures.

3.2 DISTRIBUTED COMPUTERS

A distributed computer architecture was selected as the baseline approach for building block implementation because we feel that it will have the widest range of applications. (Also, a single computer architecture is a degenerate case and is thus covered.) Since most complex systems consist of a set of subsystems, and since the availability of microcomputers is making it possible to place low cost computing where it is needed within these subsystems, we believe that there will be an ever-increasing demand for distributed computing in military applications. It has been shown in previous work (CART 77) that self-checking computers are feasible and relatively inexpensive. A distributed network of such computers can be hardware-efficient in that (1) other computers are available to aid in the repair of a faulty machine, and (2) redundancy can be provided in a selective fashion. It is felt that the high degree of modularity inherent in distributed systems best meets the varying requirements of performance and reliability, and offers the potential

for simplified fault-tolerance approaches which can be understood and thus accepted by a potential user.

A superficial view of a distributed system consists of a number of interchangeable computers connected to I/O devices through a redundant, shared busing system, as shown in Figure 3-1.

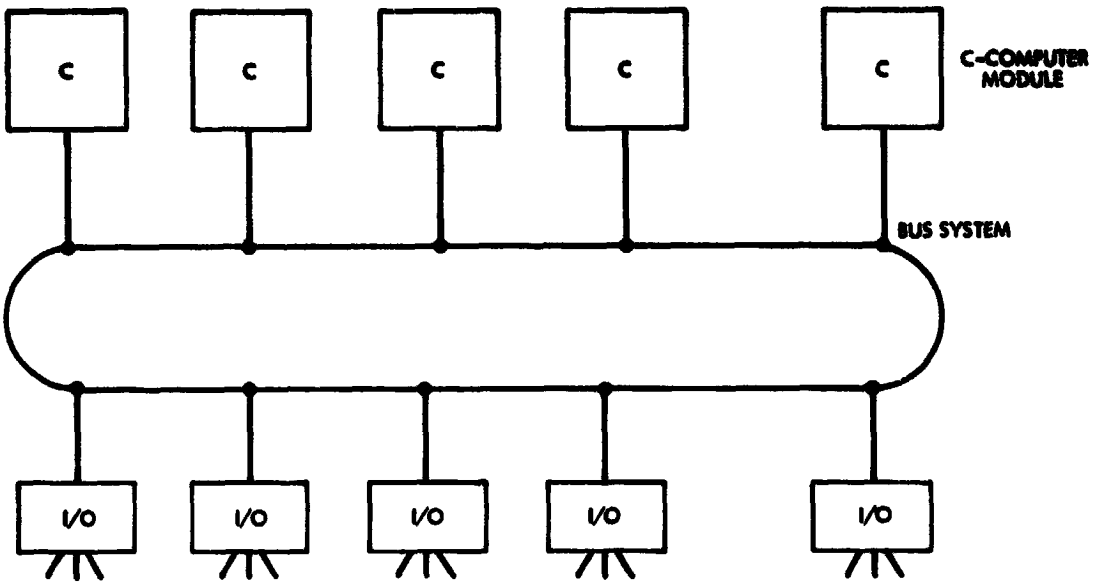


Figure 3-1. A Non-Dedicated Distributed Computer Architecture

To provide fault-tolerance, the computers may be designed with internal checking logic to detect their internal faults, or pairs of computers may run the same computations and compare outputs, or the machines may be run in triplets with output voting. A common set of backup spares is used to replace failed computers. These approaches have the advantage of nondedicated redundancy, in that any spare can be used to back up any of the active computers and a small number of spares can be used to back up a large number of active computers.

A closer look at the problem indicates that the majority of computers in such a network will be dedicated to specific subsystems. An examination of the bus interface and control logic in various subsystems shows that, for many, it is cost effective to replace the

internal control logic with a microcomputer -- either to save chips or to establish standardization in subsystem logic designs. More importantly, by establishing "intelligent" sensors and actuators through the use of local computers, system level complexity can be greatly reduced. This is seen in several ways:

(1) The subsystem-system interface can be greatly simplified, allowing the subsystem contractor to thoroughly test his device before system integration.

(2) Subsystem-peculiar computing (software) can be developed by the subsystem contractor.

(3) The computing load on central computers can be drastically reduced, since they are no longer required to generate detailed timing signals used in the associated equipment. They are instead generated in the local computer.

(4) Bus timing and loading are greatly simplified for reasons mentioned above.

Thus, the structure of distributed control systems falls rather naturally into a hierarchic structure: a large set of intelligent sensors and actuators containing their own dedicated computers, and a smaller set of non-dedicated, high-level computers which coordinate the lower level processors.

3.3 THE DISTRIBUTED COMPUTER MODEL

The model used in this report for a distributed processing architecture is shown in Figure 3-2.

Redundant elements and checking circuits are not shown in order to focus on the basic computational functions which are performed in a fault-free environment.

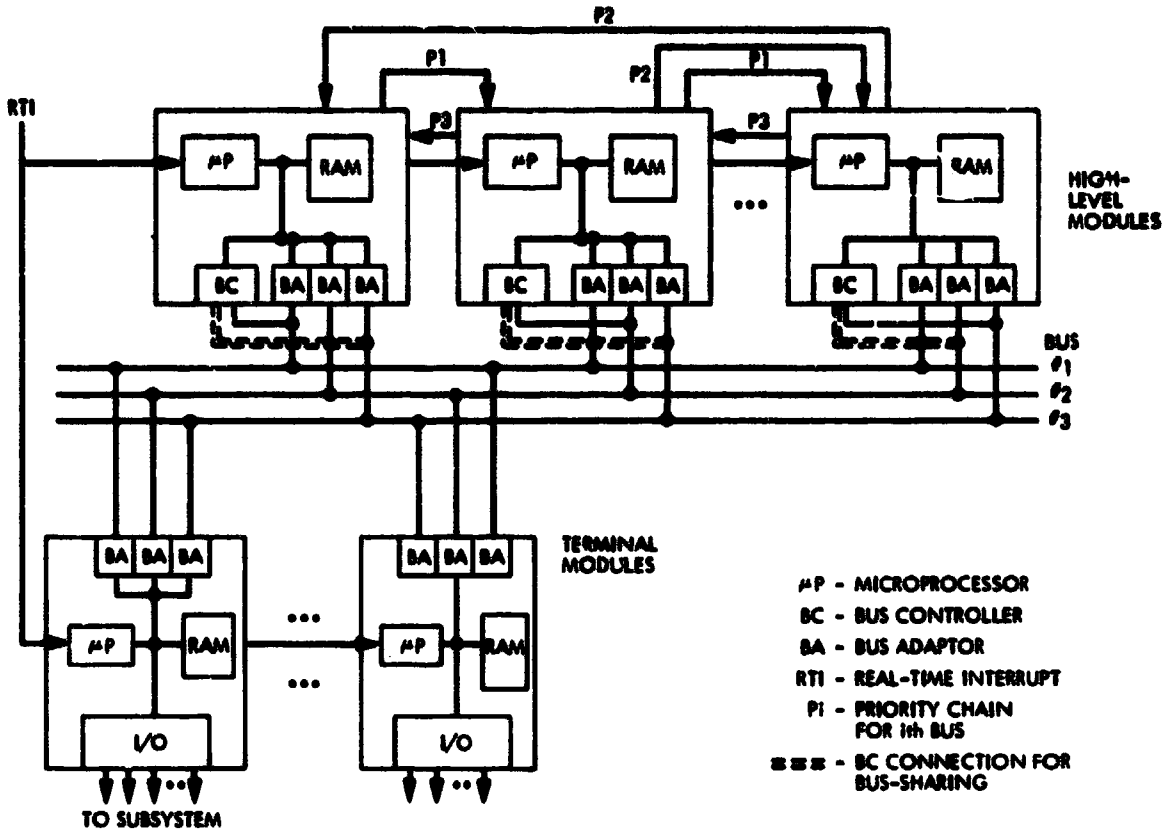


Figure 3-2. The Distributed Processing Architecture

The microcomputer modules which utilize the same microprocessor and local executive fall into two types: (1) Terminal Modules, which are configured with I/O circuits to interface with electromechanical subsystems in which they are embedded, and (2) High-Level Modules which are configured to coordinate the processing in various computers by control of an intercommunications bus.

Terminal Modules (TM) are located within the various subsystems and are responsible for local control and data collection. The Terminal Module contains a microprocessor, memory, a set of I/O modules, and a passive interface (Bus Adaptor) to each of several intercommunication buses. Each Bus Adapter contains a complete DMA controller which allows the bus system to enter or extract data from the Terminal Module's memory by cycle stealing techniques. Communication is through message slots in the local memory.

A High-Level Module enters commands, data, and timing information into prearranged memory areas within the Terminal Module. The Terminal Module delivers information to the system by placing outgoing messages in predetermined locations of its memory, which are then extracted by a High-Level Module over the bus.

The TM memory can be accessed by several buses simultaneously because the bus adaptors provide conflict resolution. The TM computer is normally not notified that data is being entered or taken from its memory. Periodic processes synchronization is provided by a common Real-Time Interrupt which triggers a local executive to check the TM memory for incoming commands and data at pre-arranged times.

High-Level Modules (HLM) are responsible for coordinating the processing which is carried out in the remote Terminal Modules or in High-Level Modules which are lower in the network hierarchy. Each High-Level Module consists of a microprocessor, memory, Bus Adaptors, and a Bus Controller.

The Bus Controller, which is unique to High-Level Modules, can move blocks of data between memories of all modules connected to its bus. Using the Bus Controller, the High-Level Module can place commands into the memories of the various computers on its bus and monitor ongoing processes by reading out selected information.

When activated, the Bus Controller reads a control table within the memory of the HLM which specifies the transfer, issues these commands over the bus to the relevant terminals in the source and acceptor modules, and then monitors bus activity as the selected modules exchange information.

Using the Bus Controller, the HLM can move a block of data from within any internal memory area of a specified source module to a specified set of contiguous locations within one or more acceptor modules.

3.3.1 The Intercommunication Bus Structure

Each active High-Level Module has a dedicated bus under its control which provides a bandwidth of approximately one megabit. In order to provide redundancy, the HLM can relinquish its bus under one of two conditions: (1) it is not powered, or (2) its processor specifically releases the bus for a specified time interval. Thus, spare modules can gain access to a bus whose processor has failed, or a bus can be multiplexed if several buses have failed.

Access to each bus by the various High-Level Modules is based on a fixed-priority assignment using a daisy chain structure, as shown in Figure 3-2, to establish this priority. Modules of higher priority, signal release of the bus via the daisy chain which then activates that hardware necessary to allow bus control by modules of lower priority. The individual buses are physically independent and, therefore, no central controller exists as a potential catastrophic failure mechanism.

The Bus Controller and Bus Adaptors are highly autonomous units which contain considerable internal microprogram sequencing to carry out their functions. For example, the Bus Controller is activated by an out-of-range store instruction in the HLM, the data "stored" is the address of a bus control table. The Controller reads out the table by DMA and controls a data transfer over its bus without further attention from the HLM processor. Completion is signalled by an interrupt with a status word stored in the HLM memory.

A bus control table in the HLM contains the identification and internal memory address of a source module, and the identification and internal addresses of one or more acceptor modules, followed by a word count. Internal addresses can be specified directly or by naming indirect pointers contained within the various source and acceptor modules. This allows accessing data by name.

The Bus Controller reads the control table and sends the source and acceptor specifications over the bus as 1553A transmit or receive commands. The source module then outputs sequential words from memory and the acceptor module ingests this data.

The Bus Adaptors contains sufficient microprogram control to recognize transmit (source) and receive (acceptor) commands directed toward their host computer. These modules then determine the base address of data to be transferred -- either a number received over the bus for direct addresses, or a number read from a mapping table in local memory for indirect addressing. The adaptors then steal cycles from the processor to transfer information into or out of its memory.

A non-fault-tolerant version of this architecture has been developed under NASA sponsorship, and a six computer breadboard has been constructed and used to verify its software and communications concepts. The breadboard was used to simulate several command, telemetry, and subsystem control functions of a planetary spacecraft. Further information can be obtained in the following references: RENN 76, LESH 76, and RENN 78b.

3.4 FAULT-TOLERANCE OPTIONS

In the distributed network, there are three distinct areas in which fault tolerance must be applied; the dedicated Terminal Modules, the nondedicated High Level Modules, and the interconnecting bus system.

3.4.1 The Terminal Modules

Since the Terminal Modules are attached by a number of wires a specific subsystem, they must have dedicated spares which are also embedded in the same subsystem. Thus, when redundancy is employed, dedicated cross-strapped redundant modules are used. This requires special short-isolated I/O circuits so that (1) a short will not disable spare modules, and (2) a faulty terminal module can be disabled and a spare module activated by simply turning off power to one and turning on power to the other.

The amount of redundancy of Terminal Modules is determined by the criticality and failure rate of an associated subsystem. For a block-redundant subsystem (i.e. two identical subsystems, primary and spare) redundant TMs may not be employed in each individual subsystem. But for a subsystem which manages a redundant set of sensors and actuators, the TM should be backed up by one or more redundant spares.

Fault detection in a TM can consist of

(1) self-checking hardware built into the computer which detects faults concurrently with normal operation.

(2) or software diagnostics for subsystems which are non-critical and can tolerate a period of erroneous computations.

The second option above is only viable if the interconnecting bus system prevents errors generated in a faulty Terminal Module from propagating through the system and affecting other modules.

Fault Recovery can be handled locally within the terminal module configuration of a subsystem or can be handled by the High-Level Modules. If fault recovery is implemented locally, TMs perform "cross checks" through their I/O logic to allow local fault detection and reconfiguration [RENN 80b]. This is often unnecessary, since the High-Level Modules provide an available intelligence which can be used for this purpose [RENN 80a]. Specifically, the Terminal Module hardware (or software, through a failed diagnostic) provides a fault indication which can be sampled over the bus by the High-Level Modules. The appropriate High-Level Module then commands reconfiguration to a backup spare via the bus. This recovery process contains a delay of a few milliseconds but is acceptable for many applications.

3.4.2 The High-Level Modules

The High-Level Modules have two salient reliability characteristics. First, they cannot be allowed to make errors, since they perform high-level control functions and can, by use of a bus, propagate damaged data throughout the network. Second, they are nondedicated and can be backed up with a common pool of spares. Two approaches were investigated for employing redundancy in High-Level Modules, voted functions and standby redundancy.

The voted functions approach consists of creating a mechanism to configure groups of three High-Level Modules to perform each separate computer function. Each triplet is voted and when a fault occurs, the

remaining two modules of an affected triplet command its replacement with a spare from the common pool [HOPK 75]. The advantage of this approach is that ongoing computations are not interrupted by a failure since the two remaining computers can continue with the ongoing computation until a convenient time to reconfigure. It has the disadvantage that it is expensive and complex. Three computers are required for each computation and the triad reconfiguration mechanism is complex, and bus bandwidth is tripled by redundant message transmissions [RENN 80b].

The standby redundancy approach uses computers which are self-checking. An HLM contains an error code protected memory, compared duplex processors, and fault-detecting bus circuitry. With a high degree of confidence, the HLM will detect its own faults when they occur. Redundant circuits are employed to disable the HLM's ability to control an intercommunication bus when a fault is detected. If the function being performed is time-critical, a backup (self-checked) module runs concurrently with the active HLM. If the primary HLM disables itself, the "hot" backup HLM springs into action, taking up the ongoing computation. For non-critical, high-level functions that can be cold-started after being lost for a second or so, no "hot" backup spare is provided. A critical function module effects its reconfiguration by activating a spare, loading it from mass storage, initializing its parameters and then restarting the non-critical process.

The standby approach is more efficient than the voted functions approach in the use of hardware, especially if some of the high-level functions do not require "hot" backup spares. The disadvantages are (1) lower "coverage" than voted approaches, and (2) time delays in recovery.

3.4.3 The Intercommunication Bus System Requirements

The intercommunication bus system should be redundant and provide restricted access so that faults are not allowed to propagate indiscriminately. Equally important, the structure and functions of the bus system directly influence the complexity and verifiability of software. Bus attributes and options that we have chosen for fault-tolerance are discussed below.

(1) **Redundant buses are required with no common failure mechanism in their assignment logic so that only one bus will fail due to any single fault. This can be achieved with a separate mechanism for each bus which assigns buses to high-level modules on the basis of fixed hardware priority. When a high-level module is disabled, its bus priority should be relinquished.**

(2) **High Level Modules should be capable of initiating bus transmissions, but Terminal Modules should be passive and not have this capability. This allows structured control and prevents a failed Terminal Module from directly upsetting the whole system. (It is expected that in many systems, some Terminal Modules will not be self-checking.)**

(3) Each high-level module should have control of only one bus for any ongoing system configuration. Centralized control is easier to verify and eliminates the indeterminate timing inherent in a multiply controlled bus.

(4) The bus structure should minimize the software complexity required for its control, and it should be used in a way that a minimum of transmissions are time-critical.

(5) The bus system should provide automatic verification of proper message transmission so that the High-Level Modules can detect faults and utilize alternate redundant buses in case of failure.

3.4.4 Architecture Selection

In order to be able to implement all of the various redundancy options (described above) we concluded that self-checking computers should be employed throughout the FTBBC architecture. Recent publications have shown that self-checking computers are feasible and can be built relatively inexpensively in VLSI logic [CART 77]. Using hardware-implemented fault detection, the self-checking computer can detect internal faults concurrent with normal operation. This property

is essential to implement standby redundancy, which is expected to be used in the majority of computers in many distributed systems. It also augments the effectiveness of voting configurations which may be employed in smaller, more critical portions of complex systems.

The Self-Checking Computer Module, and its communications interfaces are described below. This basic computer module was chosen to best meet the fault-tolerance requirements of a wide variety of potential applications.

3.5 BUILDING-BLOCK DEFINITION

The basic component of this fault-tolerant distributed computer architecture is a Self-Checking Computer Module (SCCM). The SCCM can be assembled from microprocessors and memory chips, connected by a small number of standard building block circuits described in the remainder of this chapter. Each building block is small enough to be implemented as a single VLSI chip, and provide the memory, I/O, and intercommunications functions necessary to interface the SCCM within a redundant network. The SCCMs are then used as larger building blocks in a network, in which redundant SCCMs are included to achieve fault-tolerance.

3.5.1 The Self-Checking Computer Module (SCCM)

The SCCM contains commercially available microprocessors and memories, connected by four types of building blocks, as shown in Figure 3-3. The building blocks are (1) an error detecting (and correcting) Memory Interface Building Block (MIBB), (2) a programmable Bus Interface Building Block (BIBB), (3) a Core Building Block (Core-BB), and (4) an I/O Building Block (IO-BB). A typical SCCM consists of 2 microprocessors, 24 RAMs, 1 MIBB, 3 BIBBs 2 IO-BBs, and a single Core-BB. A High Level Module is an SCCM containing an additional BIBB microprogrammed to be a Bus Controller, while a Terminal Module is a SCCM with all of its BIBBs programmed as Bus Adaptors (terminals).

The building block circuits control and interface the various processor, intercommunication, memory, and I/O functions to the SCCM's internal bus. Each building block is responsible for detecting faults

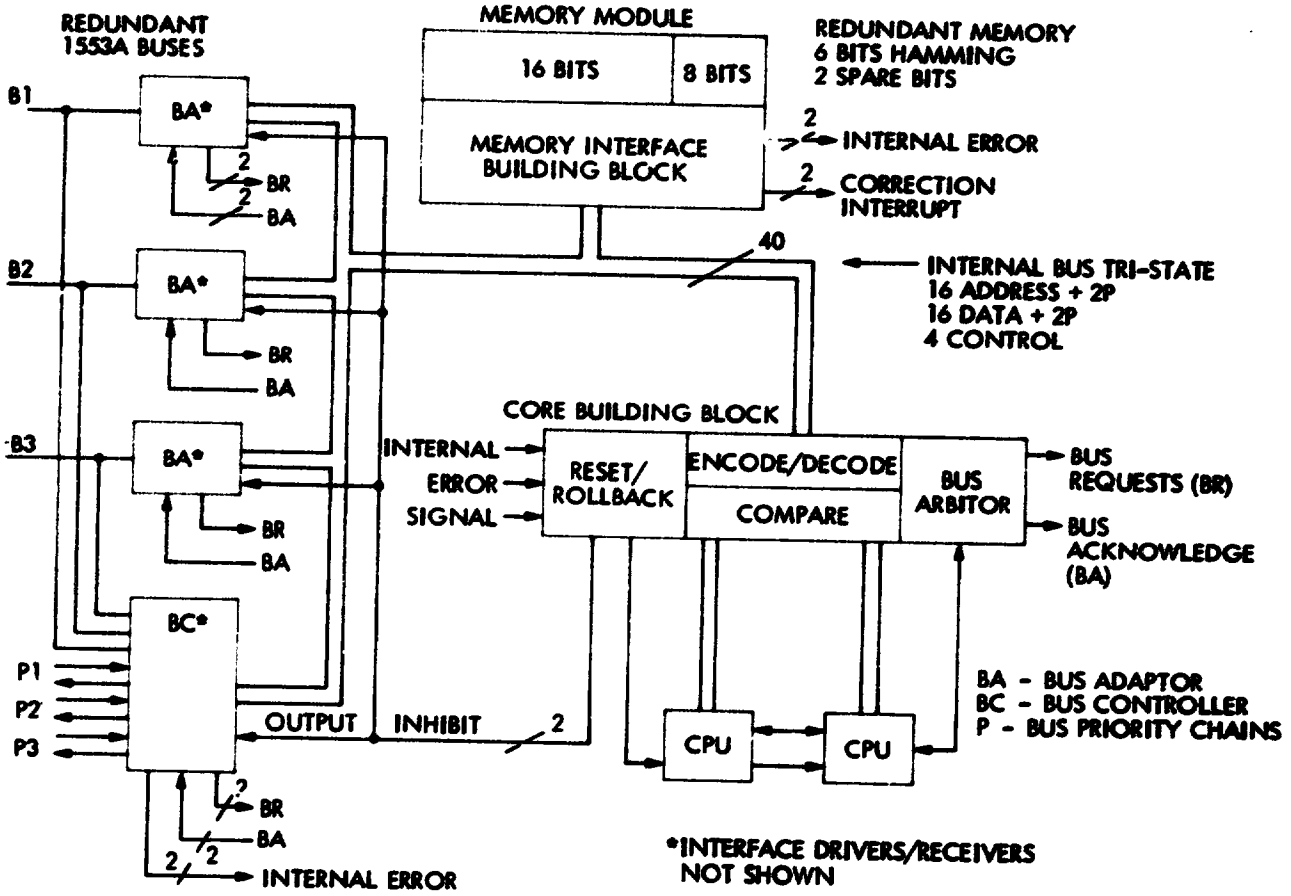


Figure 3-3. The Self-Checking High-Level Module

in its associated circuitry and then signaling the fault condition to the Core-BB by means of an internal fault indicator. The MIBB implements fault detection and correction in the memory, as well as providing detection of faults in its own internal circuitry. Similarly, the BI-BB and IO-BB provide intercommunications and I/O functions, along with detecting faults within themselves and their associated communications circuitry. The Core-BB checks the processing function by running two CPU's in synchronism and comparing their outputs. It is also responsible for fault collection and fault handling within the SCCM.

The Core-BB receives fault indicators from the other building-block circuits and also checks internal bus information for proper coding. Upon detecting an error, the Core-BB disables the external bus interface and I/O functions, isolating the SCCM from its surrounding environment. The Core-BB can either: (1) halt further processing until external intervention, or (2) attempt a rollback or restart of the processor. Repeated errors result in the disabling of the faulty SCCM by its Core-BB. Recovery can be affected by an external SCCM which is programmed to recognize the lack of activity from a faulty SCCM.

An important attribute of the building blocks is that they are interconnected via the internal processor-memory bus. They are all designed to perform specified functions in response to read or write commands to reserved addresses appearing on the internal bus. The majority of addresses are used for conventional access to RAM; however, the upper 4096 addresses are reserved for I/O functions, external bus transmission requests, the readout of error-status information, and reconfiguration commands to the building blocks. For a fetch request to a specific reserved address, the building-block circuit which recognizes the address performs the specified function and delivers a word of information to the internal data bus. Store requests to reserved addresses deliver information over the internal data bus to the selected building block. This is the commonly used technique of "memory-mapped I/O" and it has two major advantages in the building-block SCCM design. First, this approach avoids processor-specific I/O operations and thus allows

the use of a number of different off-the-shelf microprocessors in the SCCM. Second, this approach allows access to the building blocks by both software in the SCCM and from other SCCM's via the external bus system. Using the external bus an external SCCM can command DMA READ and WRITE operations into and out of the memory of the local SCCM. By directing DMA, READ, and WRITE cycles to reserved addresses, the external SCCM also has access to the building blocks in the local SCCM. The external SCCM can load and read out memory via the bus, and can also sample error status information, command internal reconfiguration, and can even remotely control I/O in a faulty local SCCM.

The following is a brief description of the building-block circuits.

3.5.2 The Memory Interface Building Block (MIBB)

The MIBB interfaces a set of RAM chips to the internal bus of the SCCM to form a Memory Module. An SCCM can contain one or more Memory Modules. A Memory Module consists of:

(1) A 24-bit memory with each bit separately packaged so that circuit failures will damage only one bit in any word. Sixteen bits are utilized for storage of computer data, six bits are employed for a SEC/DED Hamming code. The remaining two bits are used as spares to replace any of the other bits in case one fails.

(2) A Memory Interface Building Block which connects the redundant memory elements to the internal bus. The MIBB provides control, Hamming encoding and correction, spare bit replacement, parity encoding and checking for the local bus, internal checking, and error message generation.

The MIBB is connected to the SCCM internal bus and receives address, data, and two control signals: A Read/Write level, and Memory Start. Upon receiving a start command, the SCCM checks a parity coded incoming address from the bus, and for a write operation also checks

incoming data for proper coding. If no error is detected, a read or write operation is initiated and a completion signal is generated. If a single bit error is detected upon reading, it is corrected using the Hamming code.

Two fault-detection signals are generated—an internal fault indicator and a code-correction indicator. Each is sent on duplex lines so that a single fault cannot disable an indicator and go undetected.

The code-correction indicator is sent to the processors as an interrupt, and indicates that a single memory-bit error is being corrected using the Hamming Code. The processor can inspect the damaged location and, if necessary, command that the faulty bit be replaced at a convenient time.

The internal fault indicator signals all faults which cannot be corrected within the memory system. This signal is activated when:

(1) a fault is detected within the MIBB itself

(2) improperly coded information is received over the internal bus

(3) a data error occurs within the memory elements that cannot be corrected using the Hamming code.

This signal is sent to the Core building block. If the error was caused by a transient fault, correct computation can sometimes be resumed with a rollback or reset/restart sequence, initiated from the Core-BB.

The MIBB can receive several commands to read out status, test faulty locations, and perform internal reconfiguration. These commands are implemented as out-of-range memory addresses and can thus be issued by the processor or through the bus system. Specifically, certain out-of-range read or store instructions are recognized as commands to the building block and data is absorbed or disgorged for write and read operations, respectively.

MIBB commands are listed below:

(1) READ STATUS - internal fault latches are read out to the internal bus.

(2) **READ ERROR POSITION - The bit position of the most recent error is read out.**

(3) **READ ADDRESS OF LAST ERROR - The address where the last error occurred is read out to the internal bus (along with an indication if more than one bit has been corrected).**

(4) **RESET - Disables spare-bit replacement, returns to original 16-bits of data.**

(5) **DISABLE CORRECTION - Disables Hamming correction so that the memory can be externally diagnosed through the bus system under control of a different computer module. Correction is re-enabled by a reset command.**

(6) **READ REDUNDANT BITS - Used in conjunction with disable correction, reads out the Hamming protection bits and spare bit from the last address accessed in the memory.**

(7) REPLACE Ith BIT - Causes spare-bit to replace the specified bit position. (Two commands are provided - one for each spare bit plane.)

Several optional Memory Module configurations can be supported by the MIBB. The user can select the number of memory words included in the Module (8K, 16K, 32K). It is also possible to implement a Memory Module which does not use Hamming single-error correction. Using this option, each memory word consists of 16 data bits, 2 parity bits for error detection (the same code as is used on the internal bus), and 0 to 2 spare bits. Upon detection of a fault, it is necessary to diagnose the memory and command reconfiguration using an external SCCM. This option is provided for applications which require very low power, weight, and volume. Options are selected using external pins on the MIBB.

An internal error indication is generated upon receipt of improperly coded data or upon read-out of improperly coded information in the memory. The same error detecting code is employed for the internal bus and the memory plane.

3.5.3 The Core Building Block (Core-BB)

The Core Building Block provides three functions: (1) internal bus arbitration, (2) processor comparison with parity code generation and checking, and (3) fault-handling. This building block uses self-checking design, such that a fault in the Core element will result in disabling the Bus Controller and removing the module from the system.

3.5.3.1 Bus Arbitration. A Bus Arbitor in the Core-BB accepts internal Bus Request signals from the Bus Adaptors, Bus Controller and, in the case of terminal modules (to be discussed), from DMA I/O channels. Upon receiving Bus Requests, the Bus Arbitor signals the CPUs to release the bus. When the CPUs acknowledge release, the Bus Arbitor returns a Bus Acknowledge signal to the requesting element on the basis of fixed priority. Both Bus Request and Bus Release signals are duplicated with values 01, and 10 representing valid states. The Bus Arbitor is also duplicated and is compared with self-checking internal logic to detect its internal faults.

3.5.3.2 Processor Comparison, Code Generation and Checking. In order to detect processor faults, two processors are run in synchronism. Both receive the same data and execute the same programs in lock step. One processor is designated primary and the other serves as a check processor.

All outputs of the two processors to the internal bus are compared by the Core-BB and the 16-bit outputs to the address and data buses are parity encoded. Incoming data on the internal bus is checked for proper parity coding.

If the processors disagree, if incoming data is incorrectly coded, or if an internal error is detected by self-checking logic within the building block circuitry, an error message is sent to the fault-handling section of the Core-BB.

3.5.3.3 Fault-Handling. The fault-handling section of the Core building block receives internal fault signals from the various building blocks and from within the other sections of the Core. When a fault is signalled, the fault handler sends an output inhibit signal to the Bus Controller and/or IO-BBs and stops the processors. As an optional feature, the fault-handler can effect a program rollback by causing the processors to transfer to a restart location. The processors attempt to re-initialize computations. The processors can command that the module be re-enabled (release output inhibit) if no additional faults are detected in the intervening period.

3.5.3.4 Core Building Block Connections and Commands. Core Building Block Connections include:

(1) 32 address and data lines to the check processor.

(2) Control lines to and from each processor—reset/ restart, bus request for DMA, and bus released

(3) 42 connections to internal bus for address data and control

(4) Clock and Real-Time Interrupt

(5) Bus Request pairs from up to 5 DMA elements and corresponding Bus Acknowledge signals (24 lines)

(6) Internal Error inputs from up to 8 other Building Blocks (12)

(7) Output Inhibit to Bus Controller (2)

The Core-BB accepts the following commands which are decoded as out-of-range read instructions on the internal bus. Both the local CPUs and external modules can issue these commands, the latter via an external intercommunications bus.

(1) Disable Module—Computers are halted and an output inhibit is sent to the Bus Controller and/or IO-BBs.

(2) RESTART – CPUs are reset and computation begins at the rollback/restart location.

(3) Enable Module – Release output inhibit to the Bus Controller, and IO-BBs.

3.5.4 The Bus Interface Building Block (BIBB)

The BIBB can be microprogrammed as a Bus Controller (BC) or as a Bus Adaptor (BA). The bus system uses MIL-STD-1553A formats as shown in Figure 3-4, and the BC and BA provide controller and terminal functions of that standard. The capabilities of the BC and BA are augmented to provide the following additional functions:

(1) Moving data directly between memories of their host SCCMs using direct memory access (DMA).

(2) Specification of data to be moved by "name" (using automatic table look-up in the local SCCM), or by its internal memory address.

(3) Concurrent detection of message errors and faults within the BIBB. Communication of fault conditions to the host SCCM, and disabling the host SCCM under some fault conditions. Signalling SCCM shutdown via 1553A status messages.

(4) Providing redundant communications paths through the use of redundant buses.

Since a primary requirement is fault-tolerance, the BIBB is designed to detect its own internal faults. Upon detecting such an internal fault, the BC and BA behave differently. Bus Controller faults are signalled to the Core-BB which disables the host SCCM in order to prevent damaged information from propagating throughout the system. (A faulty BC can potentially move data to or from the wrong place.)

The Bus Adaptors are redundant, since several buses are connected to a given SCCM (each through a separate BA). If a BA failure does not prevent its host SCCM from performing correct computations, it is possible to re-route messages through a different BA and continue

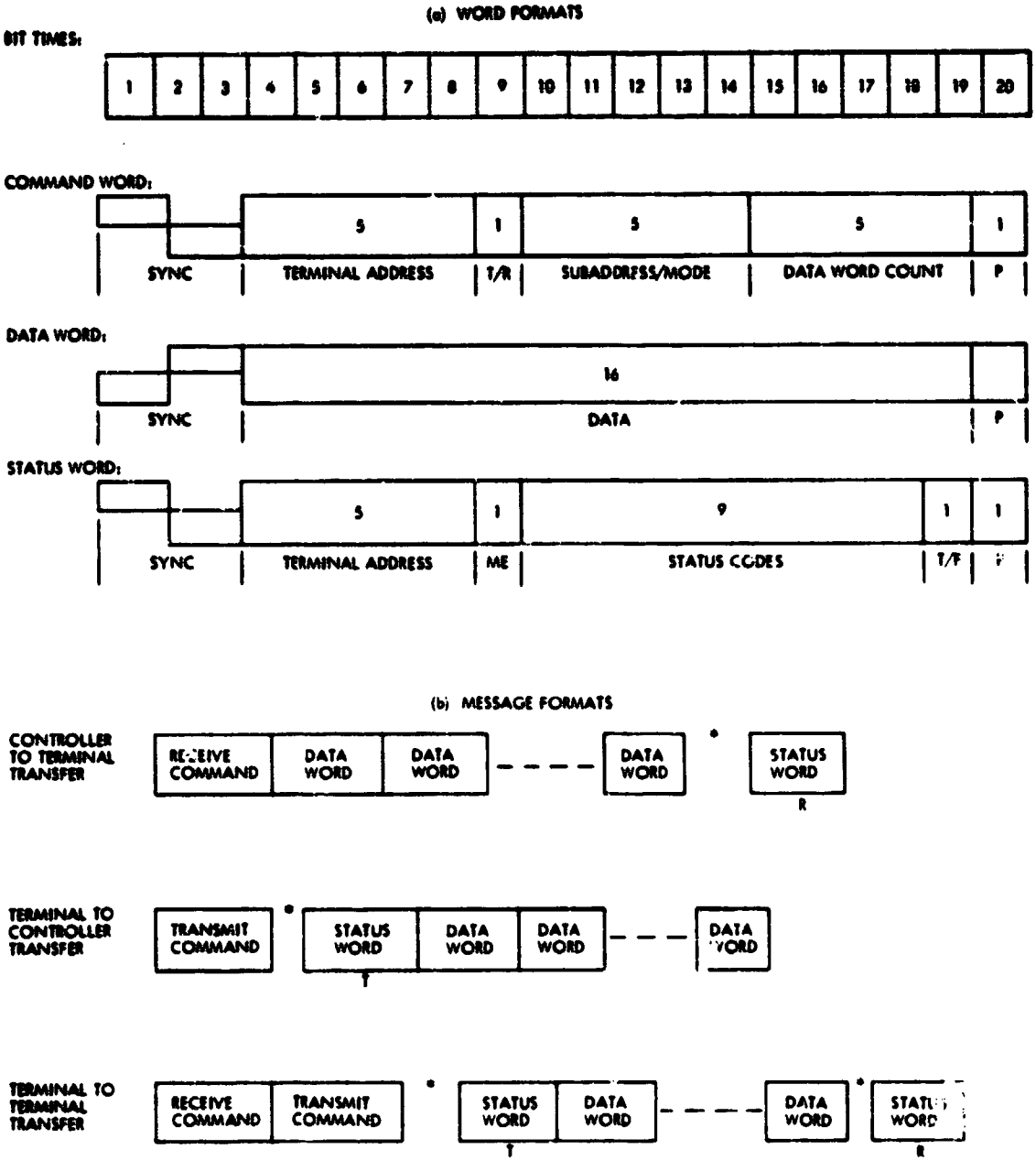


Figure 3-4. MIL-STD 1553A Formats for (a) Words and (b) Messages

normal operation. Therefore, upon detecting an internal fault, the hardware of a BA disables its ability to communicate over the external bus and into the host SCCM. It does not disable the SCCM and other BAs can be used to continue communications.

3.5.4.1 BIBB Connections and Functions. BIBB connections fall into several groups as shown in Figure 3-5.

The BIBB-SCCM Interface consists of connections to (a) the SCCMs internal address bus (AB), (b) the internal data bus (DB), (c) DMA request and acknowledge (R, AK), (d) an interrupt to the processor (RUPT), and (e) an internal fault indicator (IF). This interface allows the BIBB to enter or extract words from the local memory by cycle stealing; to alert the processor of an error or completion using the interrupt, and to signal an internal fault.

The Direct Command Interface consists of a set of output lines (DC) and a strobe signal (ST). In response to a special "direct" command, a strobe signal is delivered and the output lines can be divided to activate discrete events.

A set of Configuration Pins are hard-wired to Vcc or ground to specify the hard names of the BIBB on the 1553A external bus and on the internal bus (for memory-mapped control).

The External Bus Interface connects with discrete driver and receiver circuits for the 1553A bus. These connections include data output lines (HILO, OUTEN), data input lines ($\overline{\text{INBUS HI}}$, $\overline{\text{INBUS LO}}$), and alternate bus selection signals (BSEL, BBSY). A Bus Adaptor is only connected to a single bus. Therefore, in a BA the bus selection signals are unused. The data input and output lines are connected to a single driver/receiver package.

The Bus Controller can communicate over any of several buses. Therefore, it interfaces with a Controller Interface Module (CIM) which contains several sets of driver/receiver electronics. We have decided to place the bus assignment (allocation) logic in the CIM as well. When the BC starts to initiate a bus communication, it specifies which of

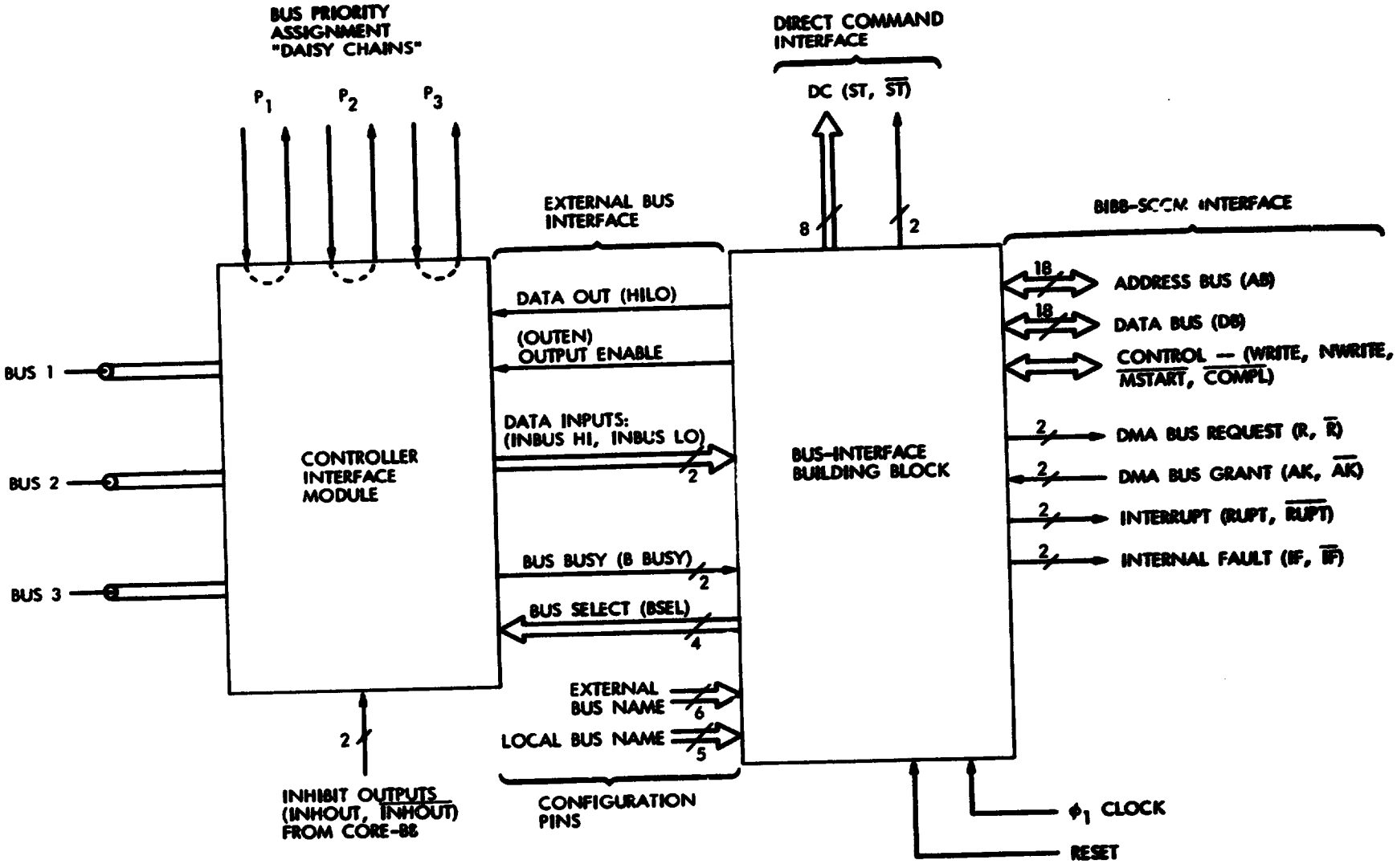


Figure 3-5. Bus Interface Building Block Connections

several buses it wishes to use (BSEL). If that bus is in use by a BC of higher priority a busy signal is returned (BBUSY).

3.5.4.2 Bus Controller. The SCCM requests its Bus Controller to execute an external bus transfer by "storing" to one of several out-of-range addresses. Four bits of this address specify which of several buses to use for the transmission. The data being "stored" specifies the address of a Bus Control Table (BCT) in memory which specifies the transmission to be carried out.

The BCT contains a control word and:

(1) One or two 1553A commands -- One for Terminal-to-Controller or Controller-to-Terminal, or two for Terminal-to-Terminal transmissions.

(2) The local address (in the BCs host SCCM) from which data is to be extracted or stored.

The BC initiates and monitors the specified transmission and moves data into or out of local memory as required. It places a BC status word in a fixed memory location and delivers an interrupt upon completion. The BC-status word indicates:

(1) Transmission Aborted, bus not available.

(2) Transmission Unsuccessful due to coding error or unreturned status.

(3) Transmission Successful but BAs SCCM has failed.

(4) Transmission OK.

(5) Activity or Requested Bus.

The status words embedded in the 1553A transmission are also stored in memory and are available for software reference.

3.5.4.3 The Bus Adaptor. The Bus Adaptor operates as an "intelligent" 1553A terminal. It is controlled via the intercommunication bus, and executes 1553A transmit and receive commands. For most commands

received over the bus, the adaptor obtains a data address corresponding to the 5-bit Subchannel/Mode (S/M) field of the command. The adaptor then deposits or withdraws words from sequential memory locations by direct memory access (DMA) to carry out the receipt or transmission of the specified number of words.

Most values of the S/M field are used as data names. These values are used as an index into a look-up table in the local memory which specifies the physical address of the named data. Several values of the S/M field are reserved for special functions. These include:

(1) Concatenate - continue extracting or depositing data from internal address used in last transmission.

(2) Designate silent acceptor - directs module to assume soft name and "listen-in" on subsequent transmission.

(3) Execute direct command - strobe data out on direct command lines.

(4) Direct addressing - specify absolute local memory address for next data to be transmitted.

The BIBB, whether programmed as a BC or a BA also recognizes several out-of-range addresses as commands to: (1) read out internal status flip flops and (2) reset itself.

SECTION 4
BUILDING-BLOCK DESCRIPTIONS

The following section presents detailed descriptions of the major building blocks. An implementation is described for the Memory Interface, Bus Interface, and Core building blocks. Each building block is broken into its component internal functions for which preliminary logic descriptions are provided. This set of descriptions was used to generate breadboard logic designs.

4.1 THE MEMORY INTERFACE BUILDING BLOCK

The fault-detecting and correcting Memory Interface Building Block (MIBB) interfaces a redundant set of memory chips to the internal bus within computer modules. It provides single bit error correction to damaged memory data, replacement of up to two faulty bit planes with spares, parity encoding and decoding to the internal bus, and detection of internal faults.

4.1.1 Memory Interface Building-Block Requirements

Memory is typically among the most significant sources of failure in computer systems. Due to the simplicity of operation and a high degree of modularity in organization, the memory system benefits most from the error-correction techniques. In particular, the application of the error correction becomes very effective in the case of semiconductor memories, organized with each bit on separate LSI chips.

The basic goal in the specification and the design of the memory building block is to provide for a highly reliable and maintainable memory system by incorporating redundancy in data representation and logic which allows thorough error detection, and correction of a majority of single-chip faults.

The fault-tolerance objective is quite simple. Since the memory represents a preponderance of failure rate within a computer module (SCCM), single fault correction in memory will greatly improve the reliability of the SCCM. Even though the SCCM is treated as a replaceable (throw-away) item with backup spares, improving memory reliability greatly increases the reliability of each SCCM and of networks made of these modules.

Specific memory interface requirements are listed below:

(1) The memory system should have the capability to correct single errors and to detect double errors in data words. This can be effectively achieved by single-error correcting, double-error detecting codes (SEC/DED codes) for the storage arrays organized using one-bit-wide memory chips (i.e., each bit of the word is located on the physically independent chip which makes all single faults affect but a single bit in a word). In order to enhance the applicability of the memory-interface building blocks, a mode with parity checking only should be provided.

(2) The memory system should be able to tolerate two faulty bit-planes in the storage array. A reconfiguration system should be provided which, upon the system command, replaces a faulty-bit plane by the spare one.

(3) Parity encode data outputs for internal (data) bus transmission.

(4) Check parity of incoming address and data off of the internal bus.

(5) Recognize Memory Interface Building Block commands as out-of-range read or write instructions. These include:

(a) Set Soft Name

(b) Read Error Status Register

(c) **Read Error Word Address**

(d) **Read Error Bit Position**

(e) **Read Check Bits**

(f) **Enable/Disable Read Retry**

(g) **Replace i-th Bit with Spare a/b**

(h) **Reset i-th Bit Replacement a/b**

(i) **Enable/Disable Single Error Correction**

(6) **Data and addresses internal to the building block shall be maintained and checked with redundant parity bits to allow internal fault detection.**

(7) **The coding and control circuits should be self-testing, fault-secure, or duplicated so that no single circuit failure will produce an undetected output error.**

(8) **A self-checked internal fault signal shall be generated (and sent to the Core building block) when a fault is detected within the Memory Interface Building Block, or when an uncorrectable error is found in memory data.**

(9) **The information about detected errors in the memory subsystem should be collected and transmitted to the system upon request, in response to the READ STATUS command.**

4.1.2 Memory Interface Building-Block Design

The memory system is organized as a random-access memory (RAM). It consists of up to 16K words of 16 data bits per word. The basic storage element is a 4K x 1 MOS static-cell chip. This chip also contains the necessary address decoding circuits, a feature essential for the error isolation and effectiveness of the error coding. The memory system operates in the conventional manner. The primary functions of the memory are to accept data, address and control information, to store that data in the location as specified by the address, and retrieve unaltered data information upon demand. The Memory System

consists of two sections; the Storage Array (SA) composed of a set of commercially available memory chips, and the Memory Interface Building Block. The Memory Interface Building Block consists of five sub-elements, designated the Address Bus Interface (ABI), the Error Control Section (ECS), the Data Bus-Storage Array Interface (DBI), and the Memory Control Section (MCS), as shown in Figure 4-1. The interface requirements, commands, structure, operation, and fault-tolerance characteristics of the storage array and the building block elements are described in the following paragraphs.

The MIBB is designed to operate in two basic modes. In HAM mode, the interface provides full error detection and correction capabilities. In $\overline{\text{HAM}}$ mode only detection via two parity bits is used. The error detection, correction and bit-plane replacement in this case are performed under the system control. The address and internal error checking remains the same in both modes. In the prototype version these modes are selected manually.

The memory size can be specified to be N = 4K, 8K or 16K words. The size is also selected manually.

Since two spare bit-modules are always provided, the storage array appears in the following configurations.

(1) In HAM mode:

16 + 6 + 2 = 24 RAM bit-planes of N bits, providing storage for 16 data bits, 6 check bits and two spare bits per storage array word.

(2) In $\overline{\text{HAM}}$ mode:

16 + 2 + 2 = 20 RAM bit-planes of N bits, providing storage for 16 data bits, two parity bits and two spare bits.

4.1.2.1 Memory-System Interface Specification. As indicated in the general diagram (Figure 4-1), the interface between the storage array and the system is achieved via the address bus, data bus and a set of control signals. These buses and control signals are specified in

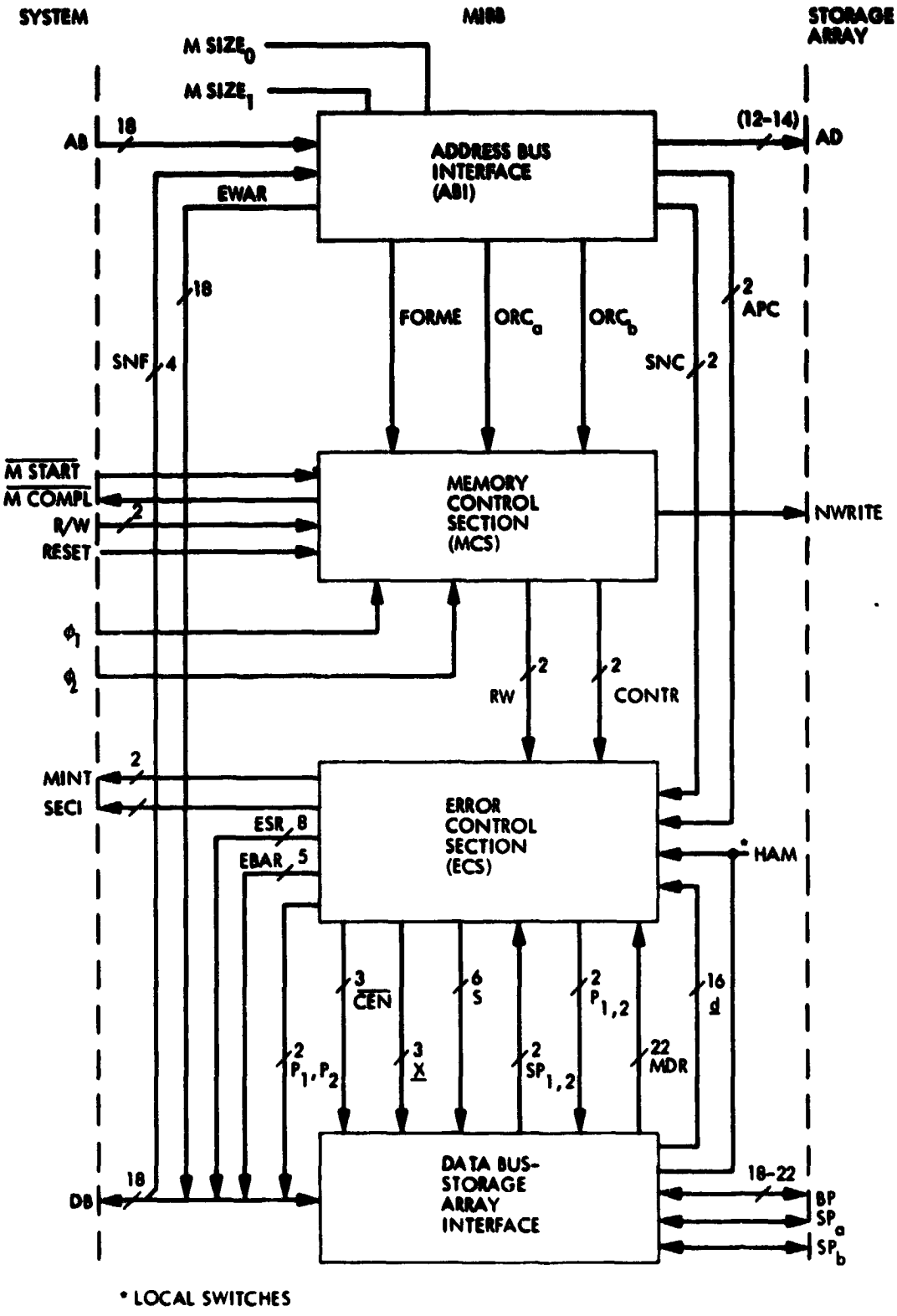
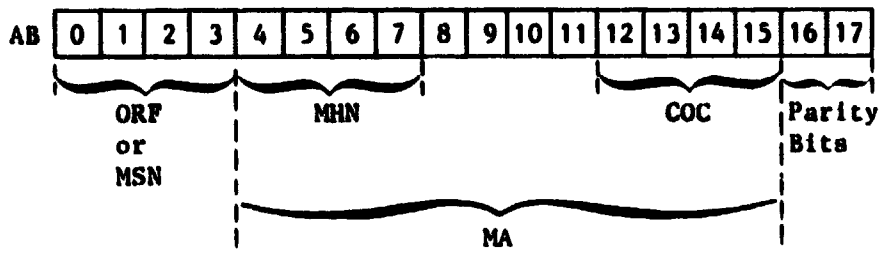
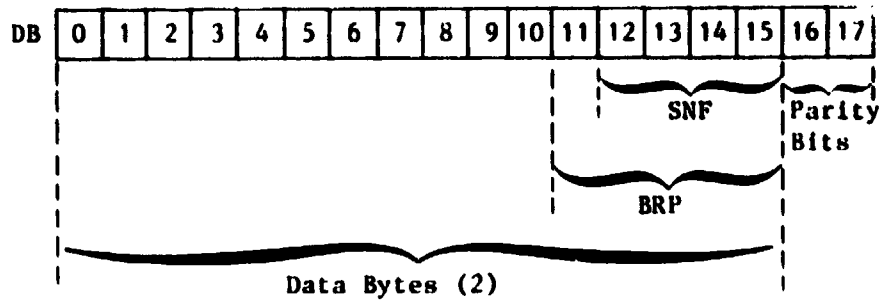


Figure 4-1. MIBB Subsystems

detail in this section. The address bus and data bus fields are indicated as follows:



(a) Address Bus Fields (for N = 4K)



(b) Data Bus Field (for N = 4K)

Address Bus

$$AB = (AB_0, AB_1, \ldots, AB_{17})$$

where AB_0 is the most significant bit and

$$AB_{16} = AB_0 \oplus AB_2 \oplus \ldots \oplus AB_{14}$$

$$AB_{17} = AB_1 \oplus AB_3 \oplus \ldots \oplus AB_{15}$$

are even and odd byte parity bits.

The address bus fields are:

Out-of-Range Field:

$$ORF = \begin{cases} (AB_0, AB_1, AB_2, AB_3) & \text{if } N = 4K \\ |(AB_0, AB_1, AB_2) & \text{if } N = 8K \\ |(AB_0, AB_1) & \text{if } N = 16K \end{cases}$$

Memory Hard Name:

- defined only if ORF = (1, ..., 1) ≡ ORC

$$MHN = \begin{cases} (AB_4, AB_5, AB_6, AB_7) & \text{if } N = 4K \\ |(AB_4, AB_5, AB_6) & \text{if } N = 8K \\ |(AB_4, AB_5) & \text{if } N = 16K \end{cases}$$

Memory Soft Name:

- defined only if ORF ≠ (1, ..., 1)

$$MSN = \begin{cases} (AB_0, AB_1, AB_2, AB_3) & \text{if } N = 4K \\ |(AB_0, AB_1, AB_2) & \text{if } N = 8K \\ |(AB_0, AB_1) & \text{if } N = 16K \end{cases}$$

Command Operation Code:

- defined if ORF = (1, ..., 1)

$$COC = (AB_{12}, \ldots, AB_{15})$$

Memory (Word) Address:

- defined if ORF ≠ (1, ..., 1)

$$MA = \begin{cases} (AB_4, \ldots, AB_{15}) & \text{if } N = 4K \\ |(AB_3, \ldots, AB_{15}) & \text{if } N = 8K \\ |(AB_2, \ldots, AB_{15}) & \text{if } N = 16K \end{cases}$$

In other words, if ORF bits are all ones then the COC bits specify a special command which is executed only by the MIBB with a physical (hard) name matching the MHN field. If ORF bits are not all ones, the MA field is used by the MIBB with a logical (soft) name matching the MSN field.

Data Bus

$$DB = (DB_0, DB_1, \ldots, DB_{17})$$

where DB is the most significant bit.

Parity Bits:

$$DB_{16} = DB_0 \oplus DB_2 \oplus \ldots DB_{14}$$

$$DB_{17} = DB_1 \oplus DB_3 \oplus \ldots DB_{15}$$

Data bytes: $(DB_0, \ldots, DB_{15})$

Soft Name Field:

$$SNF = \begin{cases} (DB_{12}, DB_{13}, DB_{14}, DB_{15}) & \text{if } N = 4K \\ (DB_{12}, DB_{13}, DB_{14}, 0) & \text{if } N = 8K \\ (DB_{12}, DB_{13}, 0, 0) & \text{if } N = 16K \end{cases}$$

This field specifies the logical name to be assigned to a memory by executing an SSN (Set Soft Name) command.

Bit Replacement Position Field:

$$BRP = (DB_{11}, \ldots, DB_{15})$$

This field specifies the position of the bit-plane to be replaced by a spare.

Storage Array Interface Signals

Memory Address:

$$AD = \begin{cases} (MAR_4, \ldots, MAR_{15}) & \text{if } N = 4K \\ (MAR_3, \ldots, MAR_{15}) & \text{if } N = 8K \\ (MAR_2, \ldots, MAR_{15}) & \text{if } N = 16K \end{cases}$$

Memory Word (Bit plane I/O)

$$BP = (BP_d, BP_c)$$

Memory Data Bits:

$$BP_d = (BP_0, \ldots, BP_{15})$$

Memory Check Bits:

$$BP_c = (BP_{16}, \ldots, BP_{21})$$

Spare Bit Plane Data:

SP_a

SP_b

Read/Write:

NWRITE

Control Signals

Memory Start:

$\overline{\text{MSTART}}$ (a 1-0 transition activates MIBB)

Memory Completion:

$\overline{\text{MCOMPL}}$ (a 1-0 transition indicates completion of an MIBB operation)

Read/Write:

RW = (WRITE, NWRITE)

= (1,0) if write

= (0,1) if read

= (0,0) if no read/write or error

= (1,1) if error

System Reset:

$\overline{\text{RESET}}$

Memory Error Interrupt:

$$\text{MINT} = \begin{cases} (0,1) \text{ v } (1,0) = 1_M & \text{- no uncorrectable memory error} \\ (0,0) \text{ v } (1,1) = 0_M & \text{- uncorrectable memory error or MINT circuit error} \end{cases}$$

Single Error Correction Indicator:

$$\text{SECI} = \begin{cases} 1_M & \text{- no single errors corrected} \\ 0_M & \text{- single error corrected or SECI circuit error} \end{cases}$$

Clock Inputs: (optional)

$\emptyset_1$, $\emptyset_2$ - standard system clocks

4.1.2.2 Specification of MIBB Operations and Commands. The commands interpreted by the MIBB are specified here as control sequences at the register-transfer (microprogramming) level in the context of the MIBB design described later in detail.

A general view of the MIBB operational states and the flow of control is indicated in Figure 4-2.

In describing commands, the following notation is used:

(1) All statements are labeled; simultaneous register transfers are separated by ";"

(2) "←" indicates register-transfer (assignment);

(3) "←" label indicates unconditional branch in control sequence;

(4) (A,B) denotes concatenation of registers A and B;

(5) All functions are implemented with combinational networks; the arguments, enclosed in ( ), are bit-vectors;

(6) For greater readability, all conditional constructs are in the form if ... then ... else ..., or if ... then.

(7) Braces "{,}" are used to enclose clauses in conditional statements.

The operations of the MIBB are specified by the following algorithms:

```
c Initialization
INIT:  if  POWER ON or RESET  then
           {ESR ← 0;          c Clear error status register
            E ← 1M;           c Clear internal error flags
            MINT ← 1M;        c Clear MIBB interrupt flag
            SECI ← 1M;        c Clear SEC flag
            EBAR ← 1;         c Set error bit-position to out-of-
                                range value
```

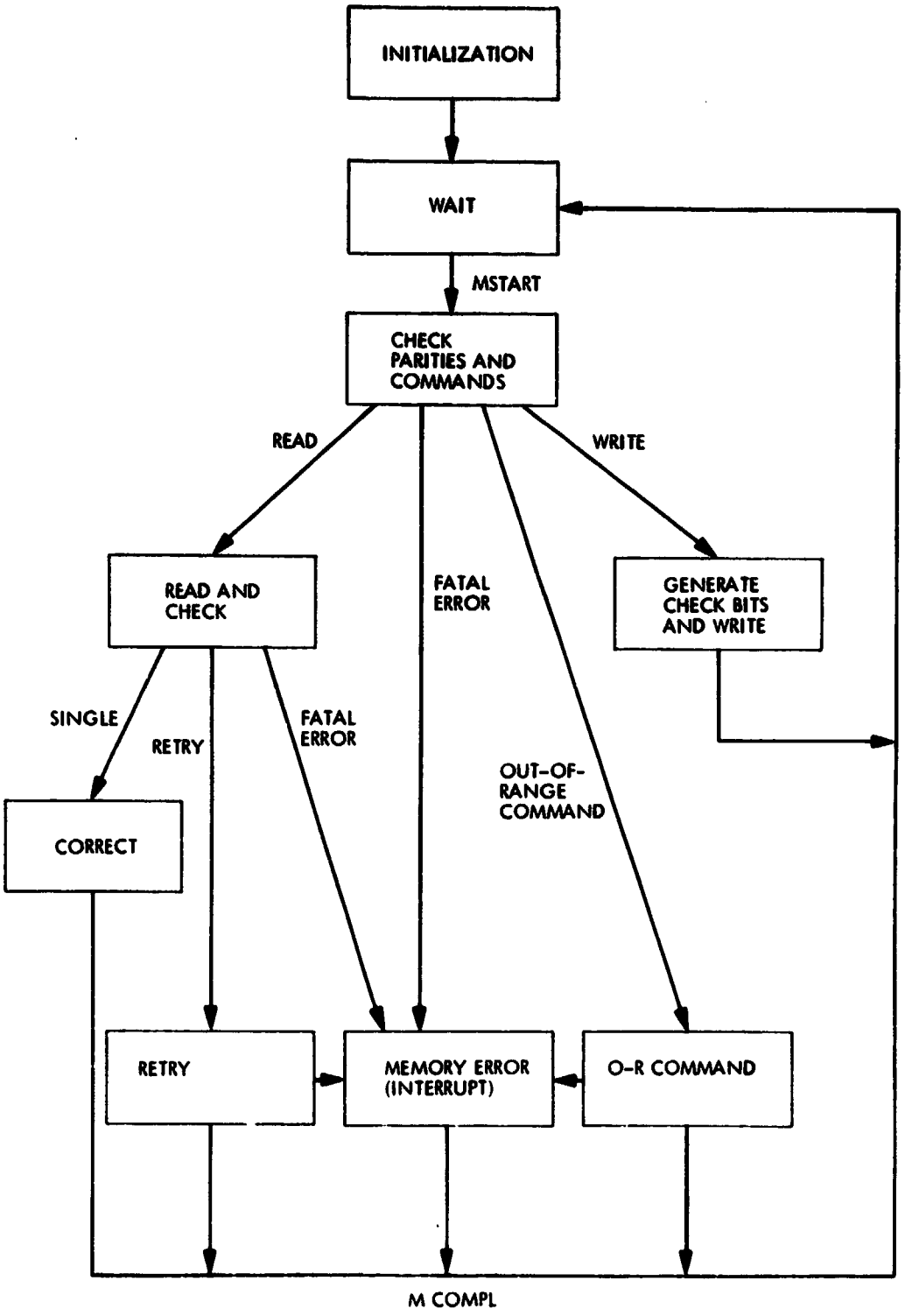


Figure 4-2. General Flow Diagram

```
                RTRY ← 0;          ¢ Clear read retry flag
                RTRONE ← 0;        ¢ Clear retry count
                FRSTEW ← 0;        ¢ Clear first error flag
                OUTEN ← 0;         ¢ Disable MIBB outputs to DB
                SECEN ← 1;         ¢ Enable SEC scheme
                RW ← 0_M;          ¢ Clear READ/WRITE command later
                POSAR_a,b ← 1;     ¢ Set bit replacement address out of
                                     range
                MCOMPL ← 1;        ¢ Memory ready
                → WAIT}

    ¢ P is 2-bit parity register; ORCR is out-of-range command
      register.

WAIT:   if  FORME•READ•MSTART  then
            {MAR ← AB; → READ1}
        if  FORME•WRITE•MSTART  then
            {MAR ← AB;   MDR ← DB;
             P ← (DB16, DB17); → WRITE1}
        if  ORC•MSTART  then
            {ORCR ← CDC; → DECODE}

    ¢ Out-of-range command decoding

DECODE:  → (decoded command)
```

READ Operation

```
    ¢ APC is address parity check; DPC is data parity check; SNC is
      soft name check;
    ¢ EWAR is error word address register which is continuously loaded
      with current address until first error; it is cleared on readout;
    ¢ EBAR is error bit position register;
```

```
¢ PAR, RED and CDR are the parity, reduction and single error
  correction functions;
¢ d and c are data and check bits from memory bit planes.
READ1:  if HAM then {MDR ← (d,c); P ← PAR(d)};
        if HAM then {MDR ← d; P ← (c0,c1)};
        ESR0,5,6 ← APC0 ⊕ APC1, SNC0 ⊕ SNC1, WRITE ⊕ NWRITE
        ¢ save Address Parity Check, Soft Name Check and Read/
          Write Check status
        E1 ← RED(APC, SNC, RW)
        ¢ reduce and save for internal use
        if FRSTEW then {EWAR ← MAR}
        ¢ store address for diagnosis
        → READ2
READ2:  ESR1,2,3,4 ← DPC0 ⊕ DPC1, SE0 ⊕ SE1, DE0 ⊕ DE1, NCE0 ⊕ NCE1
        ¢ save Data Parity Check, Single Error, Double Error and
          No Circuit Error
        E2 ← DPC;
        E4 ← RED(SE, DE, NCE);
        ¢ reduce and save
        OUTEN ← 1;   ¢ Enable MIBB data output;
                       Disable on ↓ MSTART
        → READ3
READ3:  if ERR then
        {MCOMPL ← 1; → WAIT}   ¢ No memory error; completion OK
        if HAM • SECEN • SER • NCER • RTRY then
        → READ4      ¢ Single Error correction
        if HAM • SECEN • SER • NCER • RTRY • RTRONE then
```

```
        {RW ← (0,1); MDR ← COR (w):
        ¢ read retry
        if  HAM•ERR•SER  +  HAM•E2R + RTRY•RTRONE • ERR then
        {MINT ← 0_M; MCOMPL ← 1; FRSTEW ← 1;
        → WAIT     ¢ uncorrectable memory error
READ4:  MDR ← COR(w);    ¢ Single error correction
        EBAR ← EBA;      ¢ Save bit position
        SECI ← 0_M;      ¢ Set SEC flag
        → READ 5
READ5:  P ← PAR(w);      ¢ Compute parities
        MCOMPL ← 1;      ¢ Data sent out
        → WAIT
READ6;  RTRONE ← 1;      ¢ one read retry
        BP ← w;          ¢ write back to memory
        RW ← (1,0);      ¢ switch back to read
        E ← 1_M;         ¢ reset flags
        ESR ← 0;
        SECI ← 1_M;
        MINT ← 1_M;
        → READ1
```

WRITE Operation

```
    ¢ SYN is the syndrome generation function
    ¢ w = (MDR_0, ..., MDR_15)
    ¢ No checking of single and double errors performed
WRITE1:  if  HAM  then  {MCR ← SYN(w)};
         ¢  generate syndromes
         ESR_0,1,5,6 ← APC_0 + APC_1, DPC_0 + DPC_1, SNC_0 + SNC_1,
                       NWRITE + WRITE;
```

```
          ¢ save error status
          E1 ← RED(APC: ESC  RM);
          E2 ← DPC;
          if  FRSTEW  then
             {EWAR ← MAR}
            → WRITE2:
WRITE2:   if  ERR  then
          {BP ← w};
          ¢ write only if no memory error
          if  ERR  then
          {MINT ← 0_M; ¢ no correction attempted
           ESR_2,3,4 ← SE_0 + SE_1, DE_0 + DE_1, NCE_0 + NCE_1;
           FRSTEW ← 1};
           MCOMPL ← 1;
           → WAIT
```

It is assumed that the relevant control signals, generated by duplicated controllers, are compared in each step using a morphic comparator. If an error is detected, the memory operation is terminated after setting control error status bit ESR_7 and memory interrupt indicator MINT.

The following are the "out-of-range" commands:

Set Soft Name (SSN): ¢ Duplicated soft name Registers SNR_a, SNR_b

```
SSN1:  SNR_b ← 1;          ¢ Set one register to all
                             1's to check load signals
SSN2:  SNR_a ← SNF;        ¢ Load soft name
       SNR_b ← SNF
SSN3:  if  SNC_M  then     ¢ Soft name check error
         {ESR_5 ← 1;       ¢ Set status bit
          MINT ← 0_M;}     ¢ Interrupt
          MCOMPL ← 1;      ¢ Terminate
          → WAIT
```

Read Error Status Register (RES)

```
RES1:  OUTEN ← 1;          ¢ Set connect flag
                             (reset on MSTART going low)
       MCOMPL ← 1;         ¢ Connect ESR to DB
                             (transmit error status)
       → WAIT
```

Read Error Word Address (REA)

```
REA1:  OUTEN ← 1;          ¢ Set connect flag
                             (reset on MSTART going low)
```

```
        MCOMPL ← 1;                    ¢ Connect MMR to DB
        FRSTBW ← 0;
        → WAIT
```

Read Error Bit Position (REP)

```
REP1:   OUTEN ← 1;
        MCOMPL ← 1;                    ¢ Connect MCR to DB
        → WAIT
```

Read Check Bits (RCB)

```
RCB1:   OUTEN ← 1;
        MCR ← C;
        MCOMPL ← 1;                    ¢ Connect MCR to DB
        → WAIT
```

Enable/Disable Read Retry (EDR)

```
EDR1:   RTRY ← AB11;                   ¢ AB11 = 1 to enable
        MCOMPL ← 1;                      AB11 = 0 to disable
                                              retry
        → WAIT
```

Replace i-th Bit with Spare a/b (RSP)

```
RSP1:  if  AB11  then
           {POSARa ← BRP;}   ¢ DB11,...,15 specifies
                               the i-th bit position
                               in the binary code
       if  ĀB11  then
           {POSARb ← BRP;}
       MCOMPL ← 1;
       → WAIT
```

Reset i-th Bit Replacement a/b (RBR)

```
RBR1:  if  AB11  then
           {POSARa ← 1;}     ¢ All 1's indicate a non-
                               existent bit plane.
       if  ĀB11  then
           {POSARb ← 1;}
       MCOMPL ← 1;
       → WAIT
```

Enable/Disable Single Error Correction (SEC)

```
SEC1:  SECEN ← AB11;         ¢ AB11 = 1 to enable,
       MCOMPL ← 1;             AB11 = 0 to disable
       → WAIT                  single error correction
```

4.1.3 Error Control Capabilities

The address, data and commands are systematically checked against single and double errors using appropriate encoding schemes (byte-parity, morphic and an SEC/DED code) and self-checking checkers. The information about code errors and circuit faults is collected during each memory operation cycle and saved in the error status register ESR as follows:

ESR_0 - Address Parity Error

ESR_1 - Data Parity Error

ESR_2 - Single Error

ESR_3 - Double Error

ESR_4 - Circuit Error

ESR_5 - Soft Name Decoding Error

ESR_6 - Read/Write Command Error

ESR_7 - Control Error

The error checking capabilities of the MIBB are specified in more detail next.

Address Parity Checking

$$APC = (APC_0, APC_1)$$

$$APC = \begin{cases} ((0,1) \vee (1,0) = 1_M & \text{if no parity errors in MAR} \\ ((0,0) \vee (1,1) = 0_M & \text{if parity incorrect or checker fault} \end{cases}$$

Action:

If $(APC = 0_M) \wedge (\overline{MCOMPL} = 0) \wedge CS_1$ then

$MINT \leftarrow 0_M$; $ESR_0 \leftarrow 1$; $\overline{MCOMPL} \leftarrow 1$

- no operation on the storage array performed
- CS_1 is a control state.

Data Parity Checking

$$DPC = (DPC_0, DPC_1)$$

$$DPC = \begin{cases} 1_M & \text{if no parity errors in MDR} \\ 0_M & \text{if parity incorrect or checker fault} \end{cases}$$

Action:

If $(DPC = 0_M) \wedge (\overline{MCOMPL} = 0)$ then

if WRITE $\wedge$ CS_1 then

MINT $\leftarrow 0_M$; ESR $\leftarrow 1$; $\overline{MCOMPL} \leftarrow 1$

else if READ $\wedge$ CS_2 then

MINT $\leftarrow 0_M$; $ESR_1 \leftarrow 1$

- write operation not performed on the storage array.

Data SEC/DED Checking

An odd-weight separable single error correcting and double error detecting (SEC/DED) code is used to encode 16-bit data words on 22-bit memory words [CART 76]. The SEC/DED code is specified in Table 4-1.

A memory word consists of 16 data bits followed by six check bits.

The check bits $C_0, \ldots, C_5$ are defined as

$$C_0 = \overline{\oplus} \; / \; ((MDR_{0,1,\ldots,7}) = DG_0)$$

$$C_1 = \overline{\oplus} \; / \; ((MDR_{5,6,\ldots,12}) = DG_1)$$

Table 4-1. Odd-Weighted SEC/DED Code

	Data Bits in MDR																Check Bits					
																	C_0	C_1	C_2	C_3	C_4	C_5
Bit Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Syndromes																						
S_0	1	1	1	1	1	1	1	1									1					
S_1						1	1	1	1	1	1	1	1					1				
S_2	1		1		1	1			1		1				1	1			1			
S_3	1			1			1		1			1	1	1		1				1		
S_4		1		1	1			1		1		1		1	1						1	
S_5		1	1							1	1		1	1	1	1						1
$(S_0,\ldots,S_5)$	44	35	41	38	42	56	52	50	28	19	25	22	21	7	11	13	32	16	8	4	2	1

$$C_2 = \overline{\oplus} / (MDR_{0,2,4,5,8,10,14,15}) = DG_2)$$

$$C_3 = \overline{\oplus} / (MDR_{0,3,6,8,11,12,13,15}) = DG_3)$$

$$C_4 = \overline{\oplus} / (MDR_{1,3,4,7,9,11,13,14}) = DG_4)$$

$$C_5 = \overline{\oplus} / (MDR_{1,2,9,10,12,13,14,15}) = DG_5)$$

i.e., (C_i, DG_i) has odd parity.

The check bit C_i is in MDR_{16+i}

The syndromes $S_0, \ldots, S_5$ are defined as

$$S_i = \oplus / (C_i, DG_i)$$

so that

$$S_i = \begin{cases} 1 & \text{if there is no single error in } (C_i, DG_i) \\ 0 & \text{otherwise} \end{cases}$$

The analysis of syndromes is implemented with morphic logic in the following cases:

(1) Single error:

<u>If</u> $\oplus / (S_0, \ldots, S_5) = 1$ <u>then</u>

$SE = 1_M$

i.e., an odd (actually, 3 or 1) number of syndromes with the value 1 indicates the single error case.

(2) Double error:

<u>If</u> $(\oplus / (S_0, \ldots, S_5) = 0) \wedge (S_0 \cdot S_1 \ldots S_5 = 0)$

<u>then</u> $DE = 1_M$

(i.e., a double error is indicated by an even number (< 6) of syndromes having the value 1).

(3) No single or double error

If $S_0 \cdot S_1 \ldots S_5 = 1$ then

$NE = 1_M$

No error case is indicated when all syndromes have the value 1.

Memory Control and Command Checking

The read/write command is checked by morphic logic and error causes $ESR_6 \leftarrow 1$, $MINT \leftarrow 0_M$ and no action on the storage array. The out-of-range commands are implemented using two micro-programmed control units, checked with morphic comparators at the control signal outputs. In case of discrepancy, $ESR_7 \leftarrow 1$, $MINT \leftarrow 0_M$ at the operation is terminated. The memory name decoding is checked by duplication and morphic comparators. All checker circuits are checked using morphic logic against single errors.

4.1.4 Design of Memory Interface Building Block

As indicated in Section 4.1.2, the memory system consists of two sections: the Storage Array (SA) composed of a set of commercially available memory chips, and the Memory Interface Building Block.

The Storage Array consists of up to 22 active bit-planes, denoted BP_i, $i = 0,1,\ldots,21$, which are used for storing 16 data bits and six check bits. The check bits are defined by a modified Hamming SEC/DED code for which relatively efficient implementation with good coverage can be specified. There are two spare bit-planes, SP_a and SP_b.

All bit-planes are identical and contain up to 4 (4K x 1) basic memory chips with on-chip decoding. The reconfiguration is performed by replacing a faulty-bit plane using a direct spare demultiplexing replacement scheme, as described later.

The Memory Interface Building Block is partitioned in four sections (see Figure 4-1) which are described in detail in the following paragraphs.

4.1.4.1 Address Bus Interface. The Address Bus Interface (ABI) section, which provides the address parity checking and decoding required to select a memory module, is shown in Figure 4-3. Specifically, it receives a 16-bit memory address encoded with two byte-parity bits from the Address Bus and stores it into the Memory Address Register (MAR). The self-checking parity checker circuit (APC) is used to validate the address before a read or write operation is performed. If no errors are detected, the low-order 12 bits are sent to the Storage Array Block where the independent, on-chip decoding is performed. A fault in one on-chip decoder may cause access to a wrong location to occur, but this will be detected and corrected by the data-word SEC/DED code. Similarly, two decoding errors will be detected by the SEC/DED scheme. No distinction is made between errors caused by faults in on-chip decoders or storage cells.

The decoding of the high-order (0-2) address bits, which are used to select a module within the Storage Array, are checked by a self-testing decoder. Alternately, a separate decoder can be associated with each bit-plane, thus making it possible to use the data-word error code for correction of single bit errors and detection of double bit errors in the address decoding. The high-order, module select bits are used as "soft" names and must be mapped into the physical module address.

The design of the Soft Name Checker (SNC) is given in Figure 4-4. The Address Parity Checker and the 5-input morphic comparator are shown in Figures 4-5 and 4-6. The Error Word Address Register (EWAR) is used to store the address currently being referenced. If a fault should occur the EWAR can be read out for subsequent diagnosis. The block labeled ORC detects out of range commands to the MIBB. It is shown in Figure 4-6.

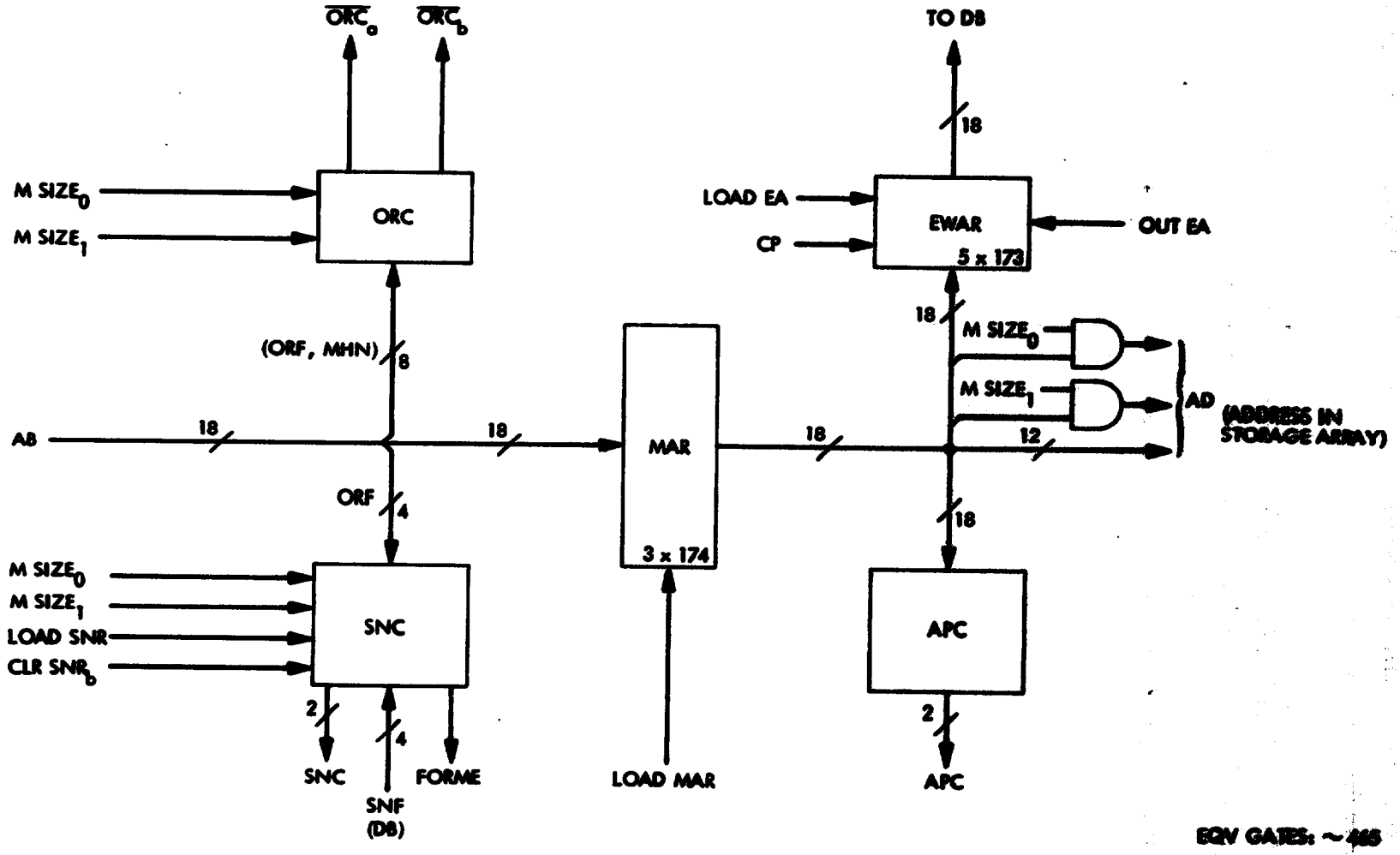


Figure 4-3. Address Bus Interface

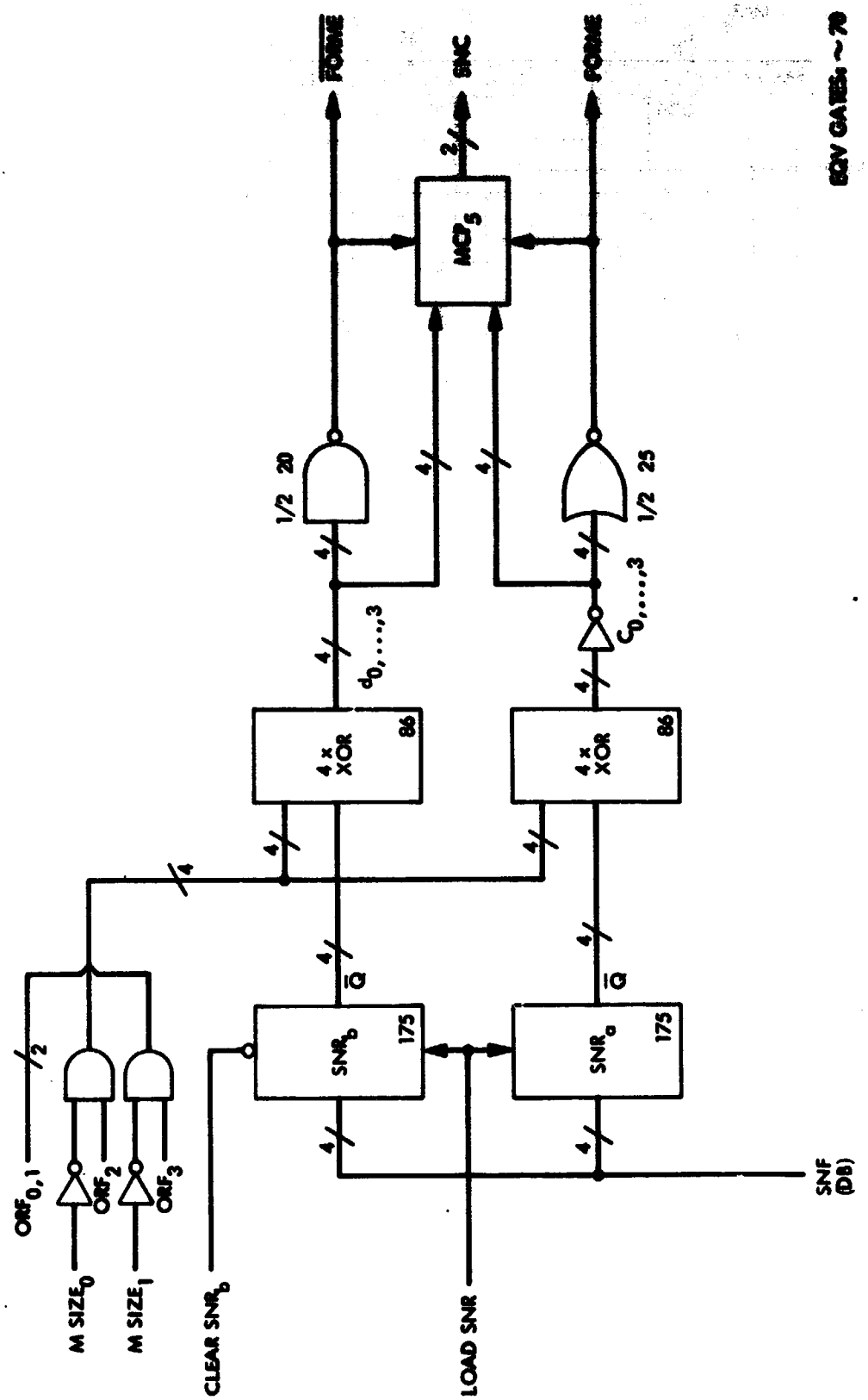


Figure 4-4. Soft Name Checker (SNC)

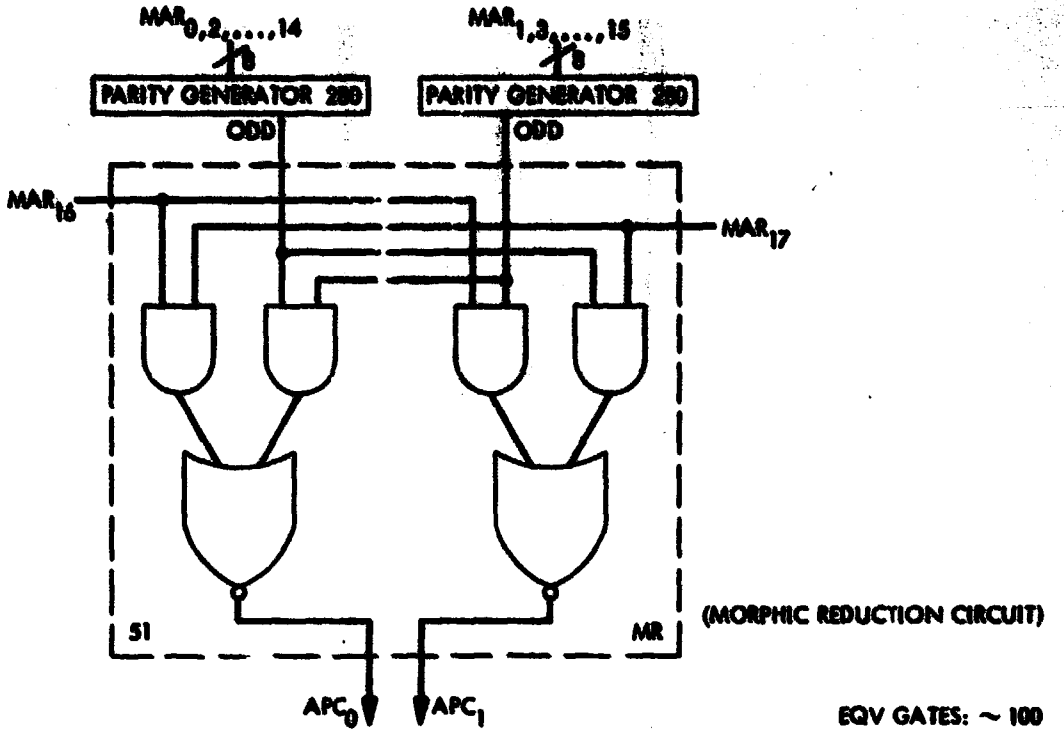


Figure 4-5. Address Parity Checker (APC)

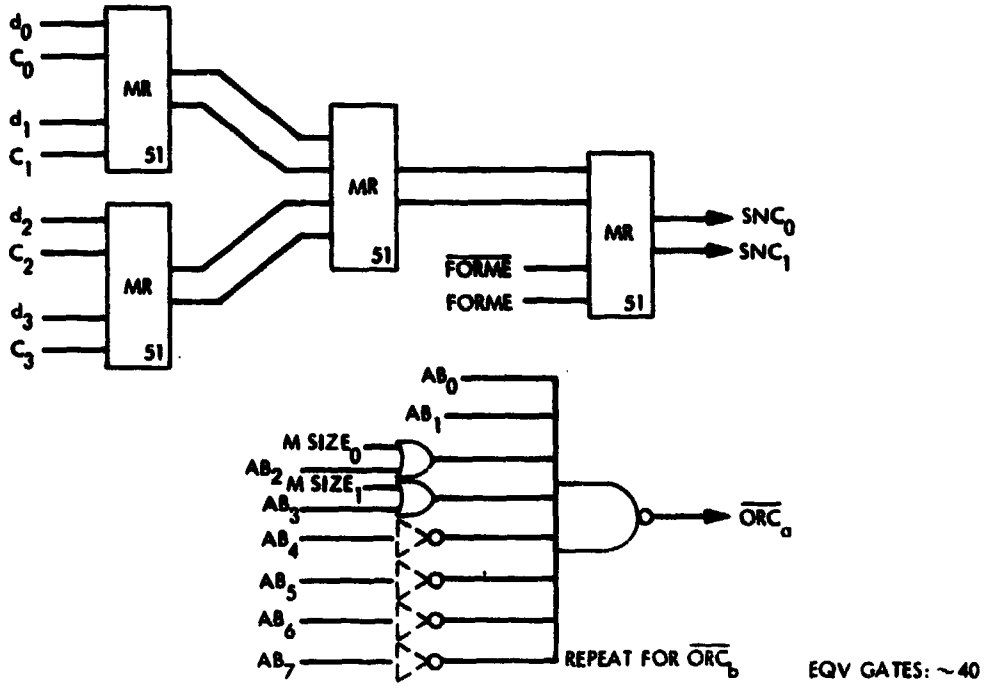


Figure 4-6. 5-Input Morphic Comparator (MPC_5)

4.1.4.2 Data Bus - Storage Array Interface. The Data Bus - Storage Array Interface (DBI) consists of three Data/Check Bit modules (DBM, CBM) and the Replacement Control Section (RCS), as shown in Figure 4-7. A Data Bit Module (DBM) is 8-bit wide. It consists of a portion of the Memory Data Register (MDR) and networks for interfacing MDR with regular and spare bit-planes. The designs of subblocks are indicated in Figures 4-8 to 4-13. The replacement of a faulty bit-plane is done by decoding replacement registers $POSAR_a$ ($POSAR_b$) (Figure 4-14). The decision which bit-plane to replace is made by the system. On the basis of error information (location of last faulty bit), the system sends the corresponding RSP command and loads $POSAR_a$ ($POSAR_b$) with the bit-plane position code. A correction input is used to allow the error correction subsystem to complement an erroneous bit. The concurrence of POS_a and $\overline{EN}_{ai}$ causes the specified bit to be replaced by spare-a in the ith DBM or CBM. Similarly $\overline{EN}_{bi}$ enables replacement of the internal bit specified by POS_b using spare plane S_b. The signal $\overline{CEN}$ enables correction (inversion) of the bit specified by $X_{0,1,2}$ within the selected DBM (or CBM). The signals S_0-S_5 are the Hamming code syndromes to be inserted in the check bits during store operations.

4.1.4.3 Error Control Section. The Error Control Section (ECS), shown in Figure 4-15 is responsible for generating Hamming code check bits and syndromes (SGC) (see Figure 4-17), byte-parity generation and checking (DPCG) (see Figure 4-16), and error analysis (SDA). The circuits used in ECS block are also self-testing. The single bit error is corrected by a decoding syndrome generated from the word contained in the Memory Data Register (MDR) in order to localize the faulty bit i. The correction is performed by reloading MDR_i with the faulty bit complemented. The correction mechanism can be disabled on system request to preserve the data information for systems diagnostics. The byte-parity checking provides for detection of most frequent errors in the bus and interface circuits.

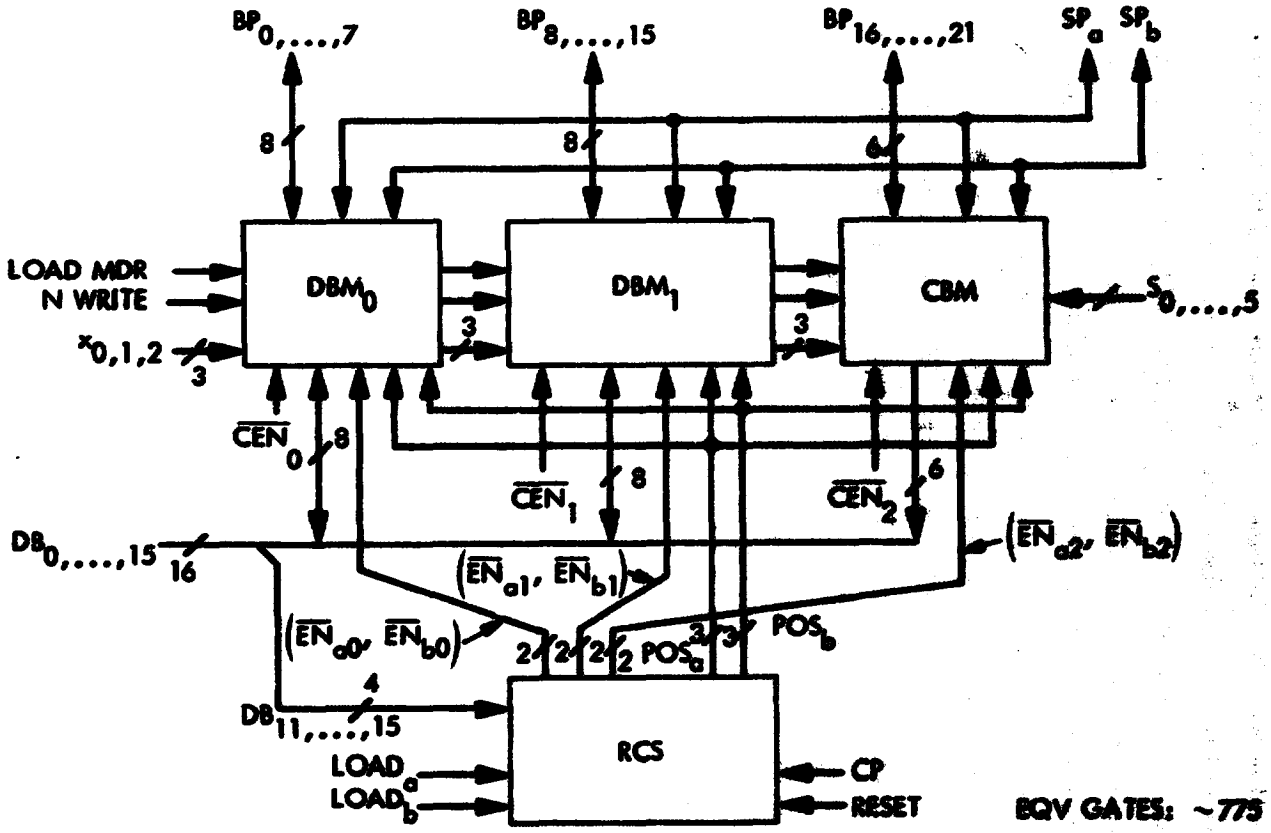


Figure 4-7. Data Bus-Storage Array Interface

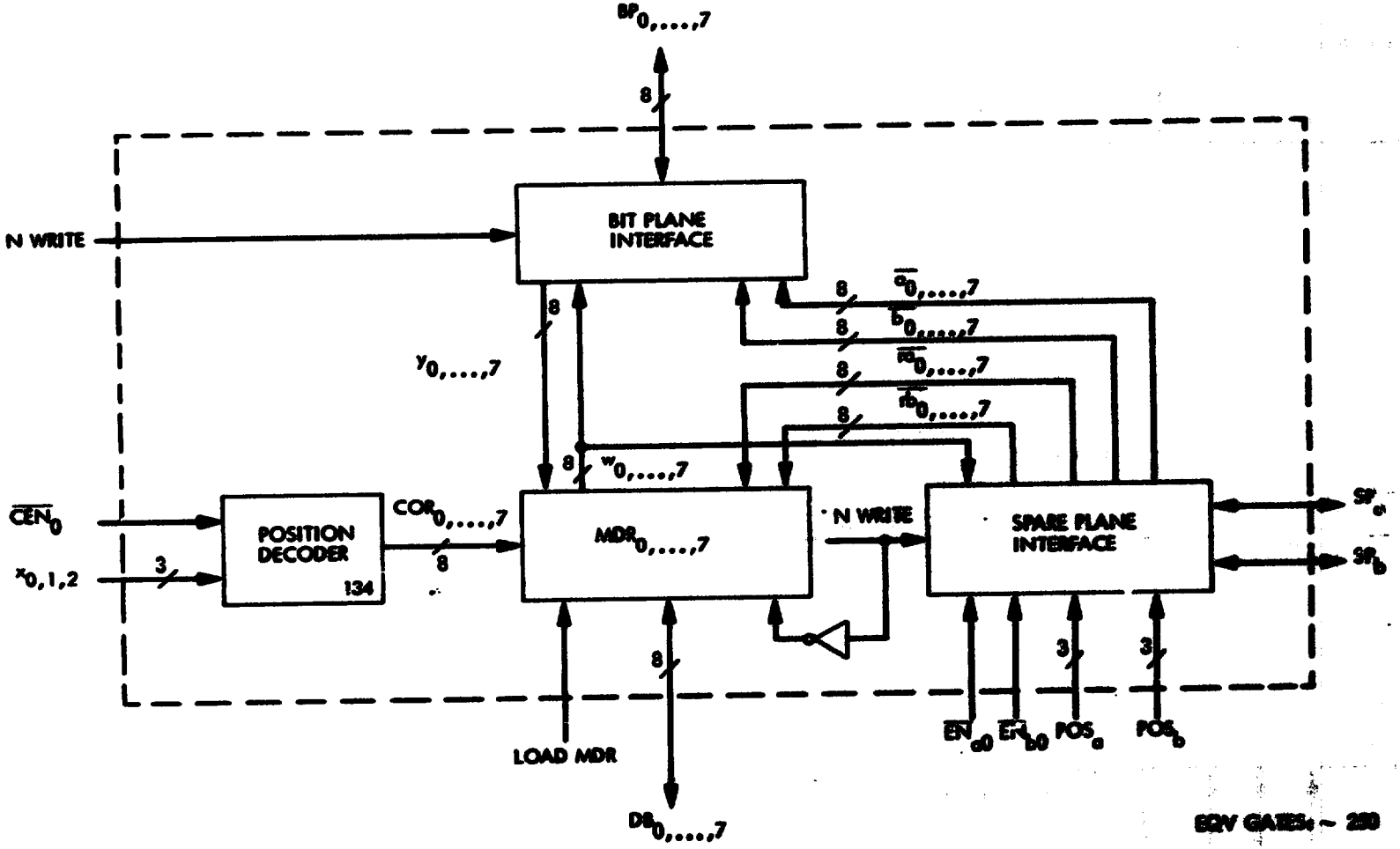


Figure 4-8. Data/Check Bit Module (DBM, CBM)

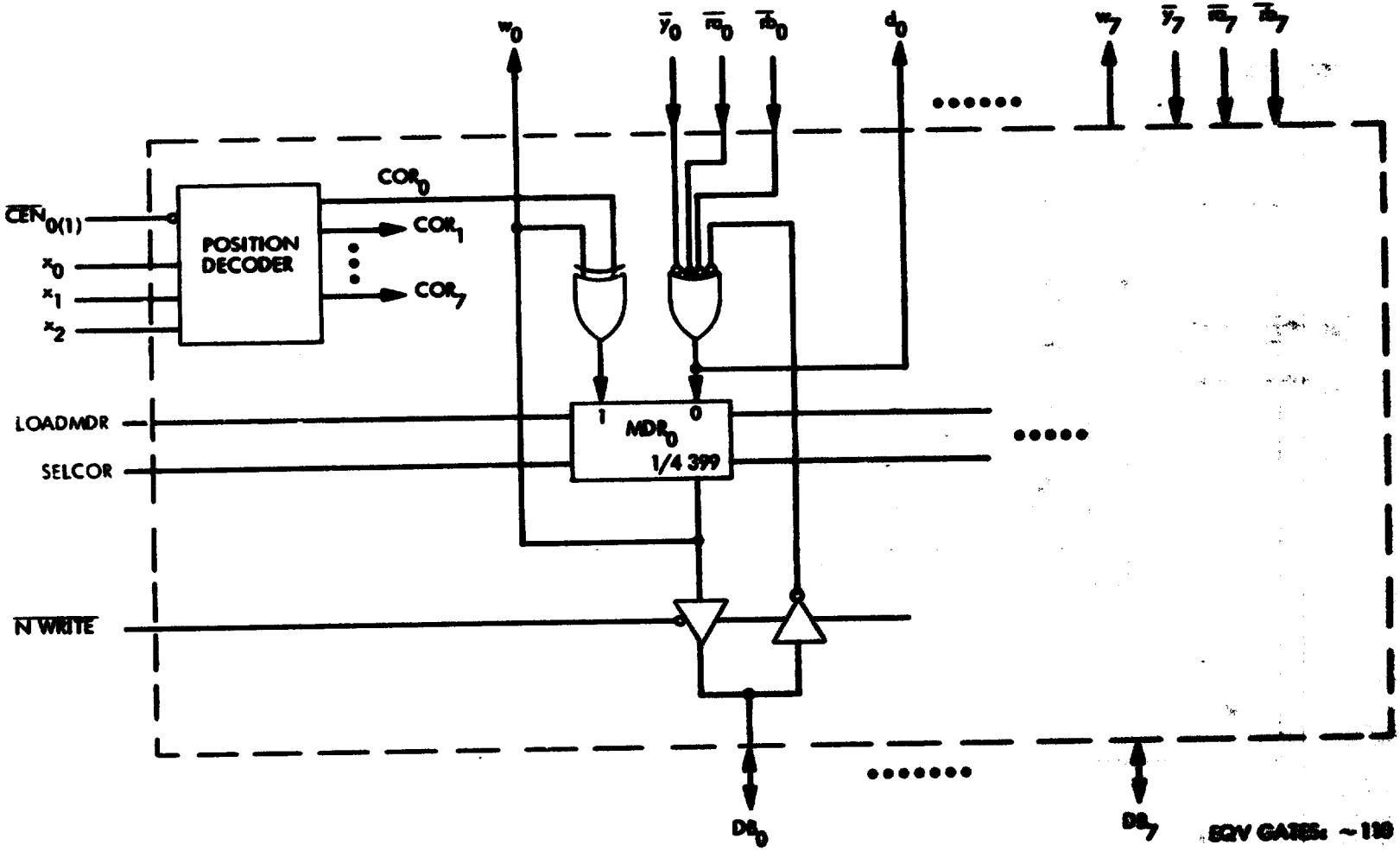


Figure 4-9. Memory Data Register - Data Bit Module (2X)

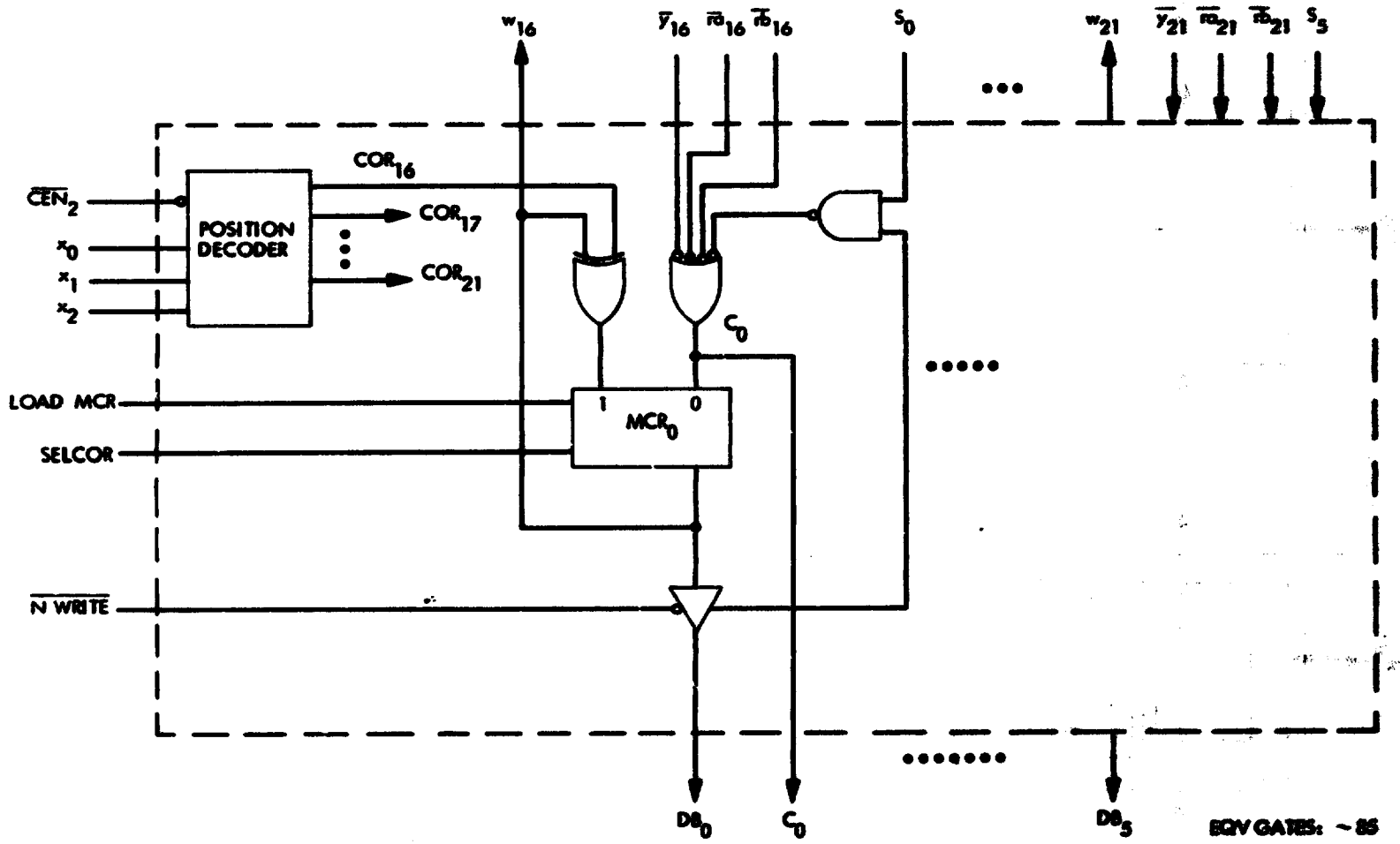


Figure 4-10. Memory Data Register - Check Bit Module

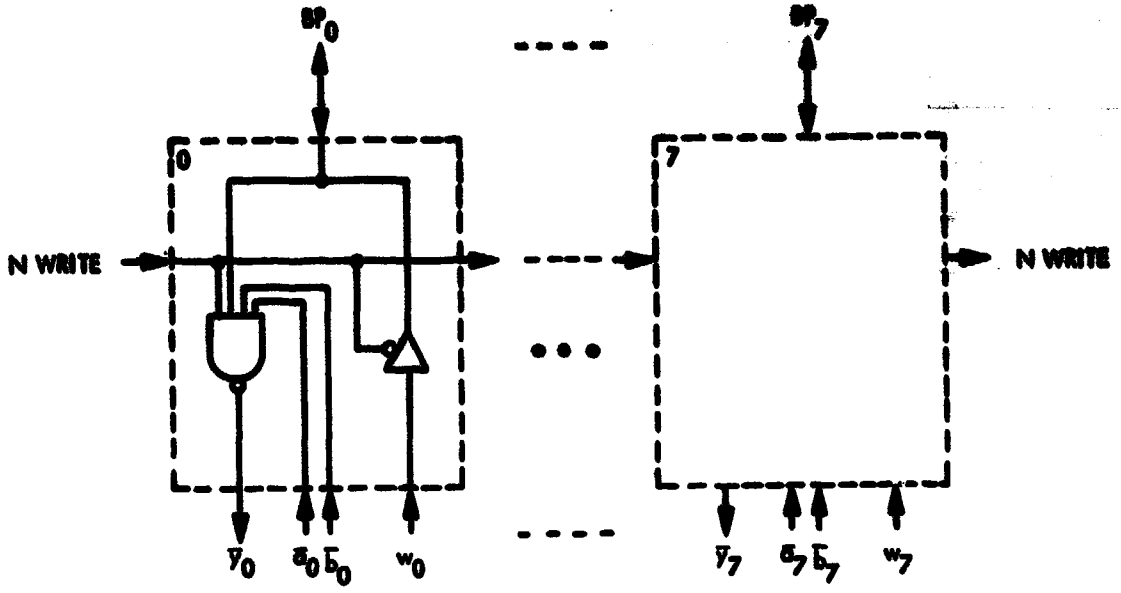


Figure 4-11. Bit Interface Module (3X)

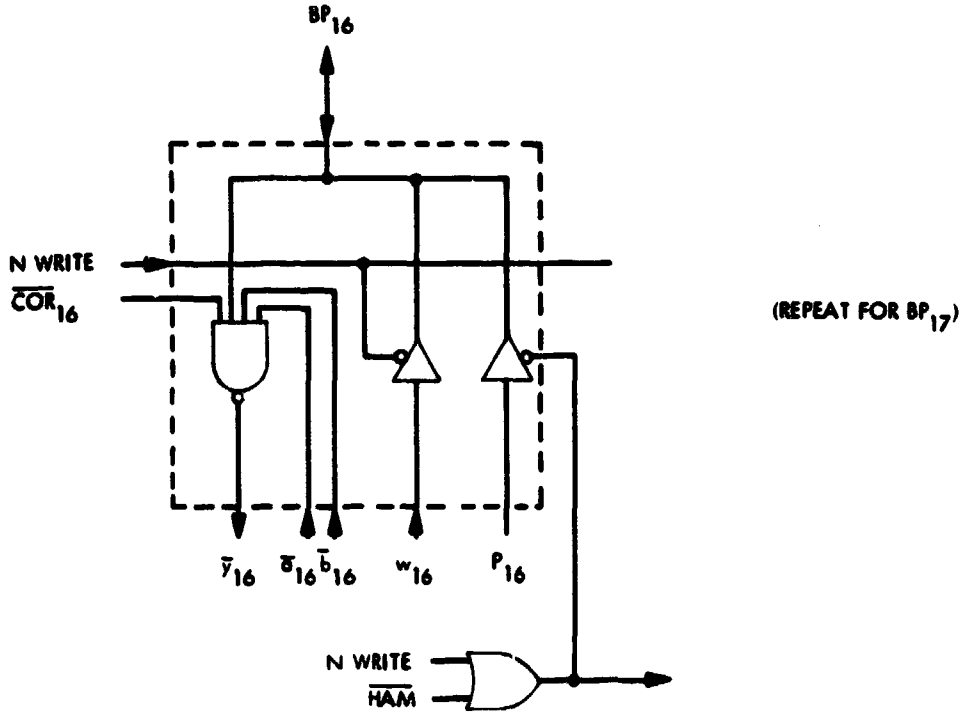


Figure 4-12. Bit-Plane Interface Module (2X)

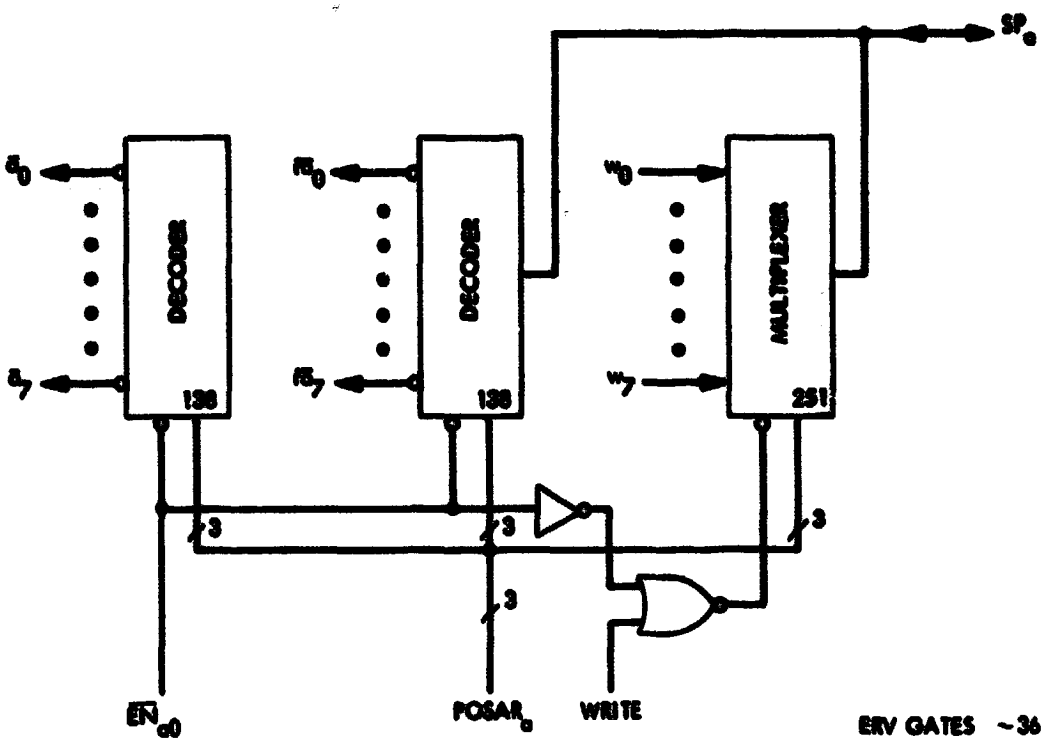


Figure 4-13. Spare Plane Interface Module (3X for SP_a, 3X for SP_b)

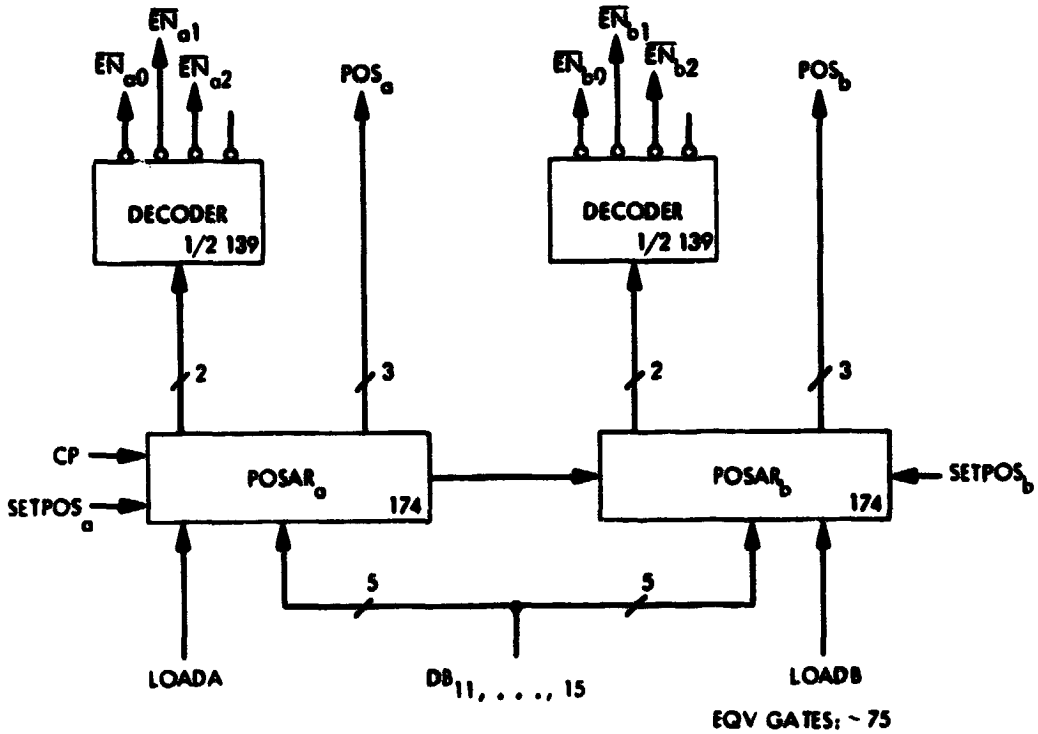
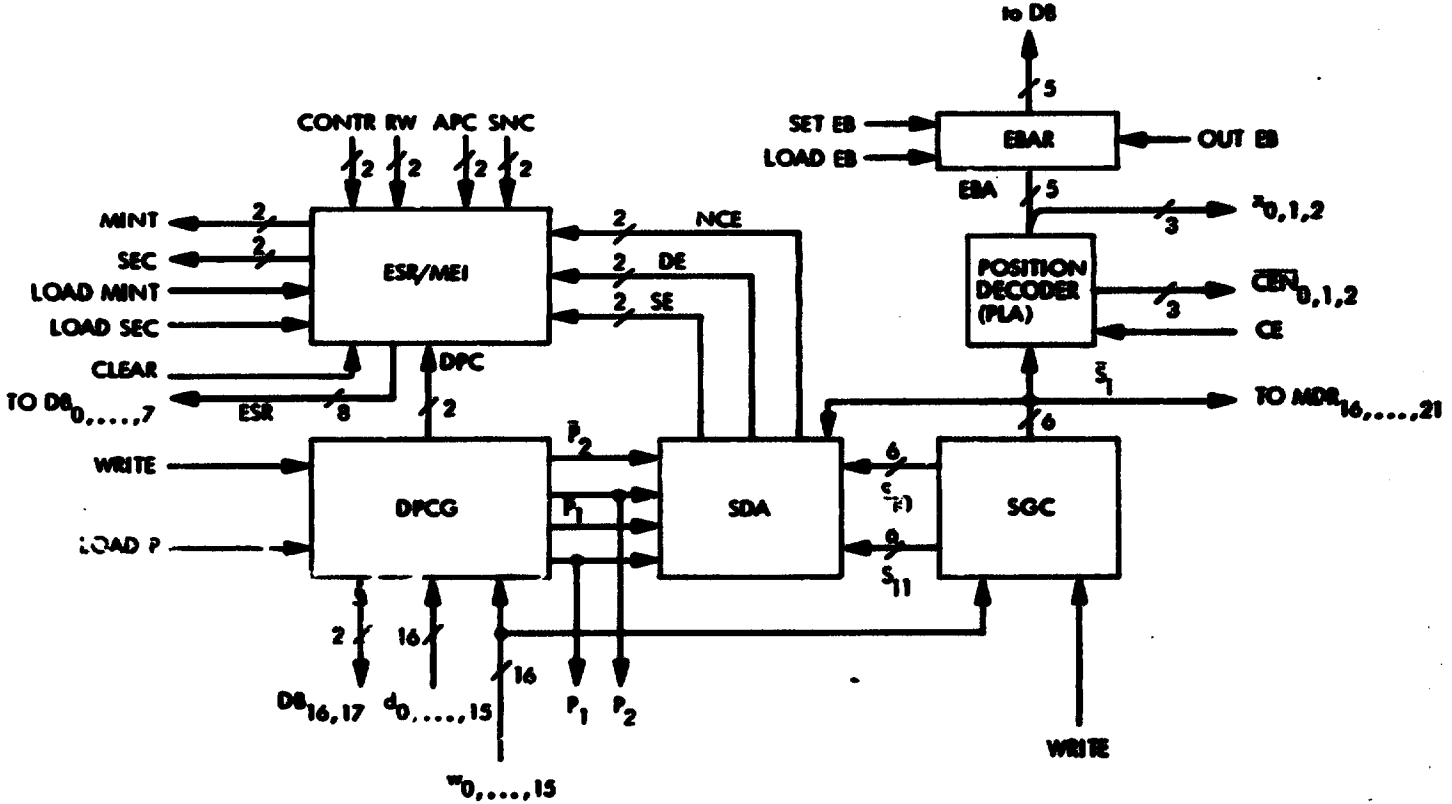


Figure 4-14. Replacement Control Section (RCS)



ESR/MET - ERROR STATUS REGISTER AND MEMORY ERROR INTERRUPTS
DPCG - DATA PARITY CHECKER/GENERATOR
SGC - SYNDROME GENERATORS/CHECKERS
SDA - SINGLE- AND DOUBLE-ERROR ANALYZER

EQV GATES: ~650

Figure 4-15. Error Control Section (ECS)

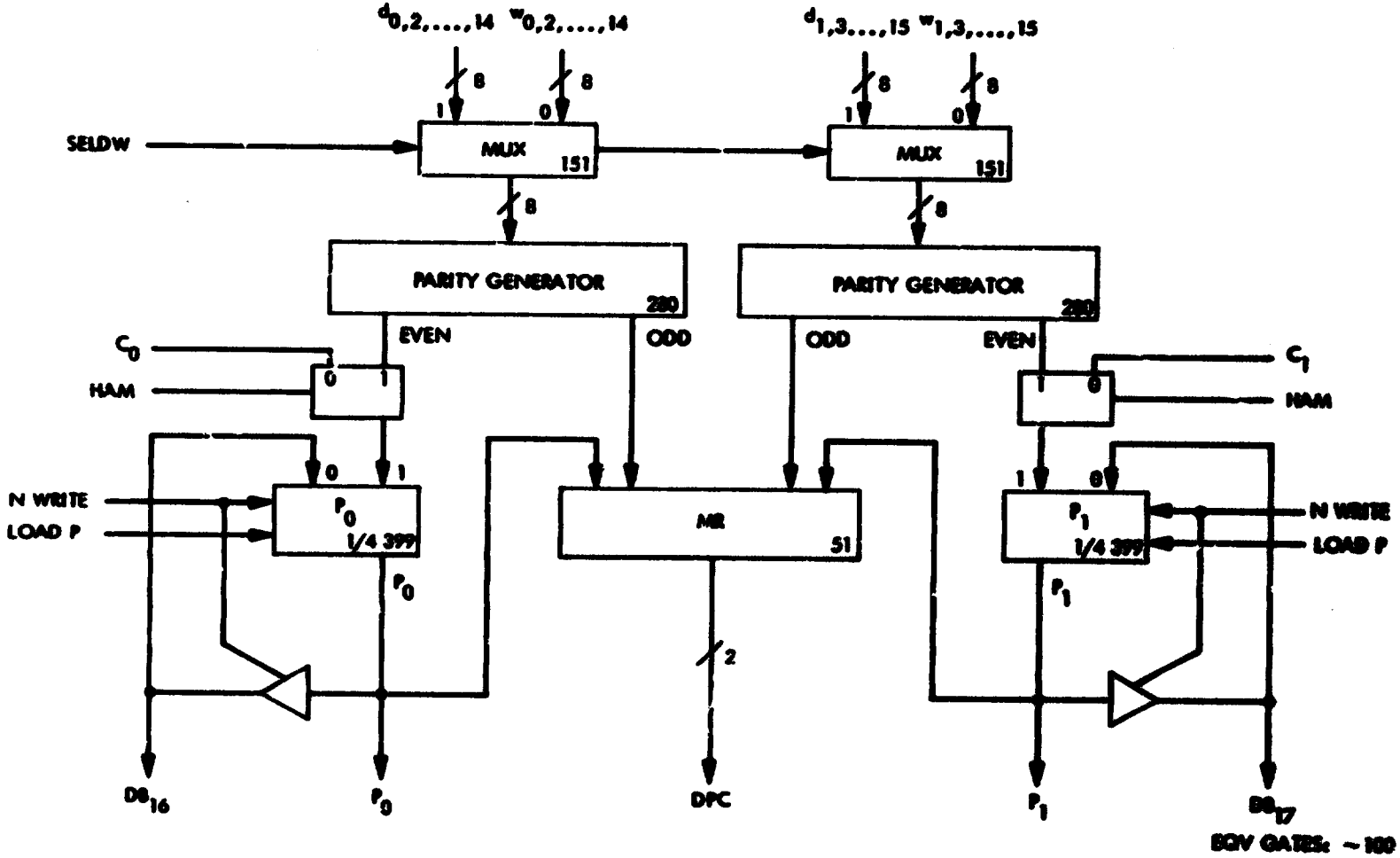
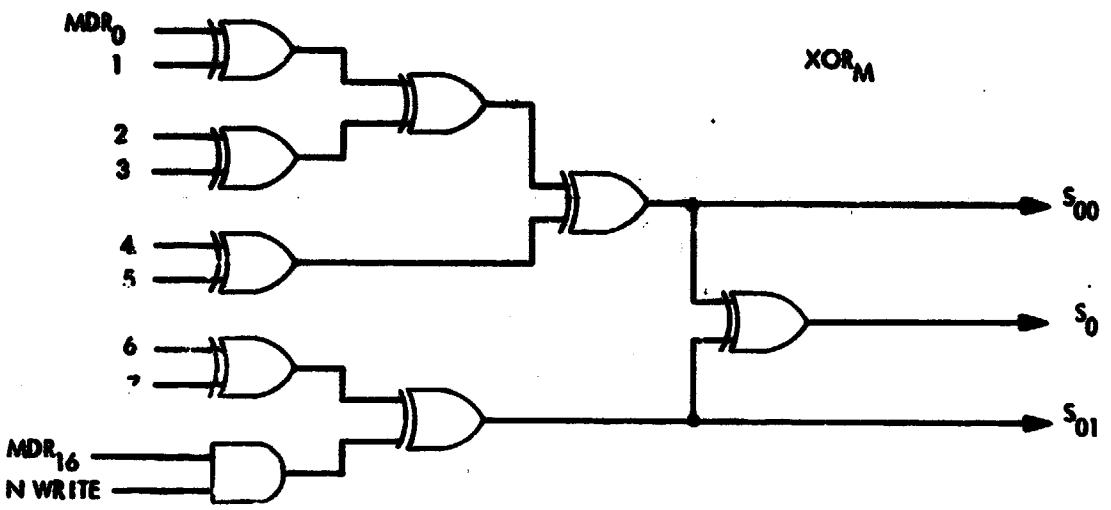


Figure 4-16. Data Parity Checker-Generator (DPCG)



REPEAT FOR	INPUT
(S_{10}, S_{11})	MDR (5,6,7,8,9,10,11,12,17)
(S_{20}, S_{21})	MDR (0,2,4,5,8,10,14,15,18)
(S_{30}, S_{31})	MDR (0,3,6,8,11,12,13,15,19)
(S_{40}, S_{41})	MDR (1,3,4,7,9,11,13,14,20)
(S_{50}, S_{51})	MDR (1,2,9,10,12,13,14,15,21)

EQV GATES: ~ 150 (TOTAL)

Figure 4-17. Syndrome Generators/Checkers (SGC) 6X

The error analyzer receives the inputs from the following functions: data-word error coding; data-word byte-parity checking; address-word byte-parity checking; all self-testing circuits and checkers of duplicated units. The output signals indicate the conditions, such as NE (no error), SE (single error), DE (double error), CE (circuit error), and they are recorded in the Error Status Register (ESR) which can be transmitted over the Data Bus on system demand. The specification of the fields and information to be recorded in ESR should enhance the systems diagnostics and maintainability of the memory system.

The design of ECS follows that of Carter et al.

The morphic XOR trees are used in checking and generating check bits as follows:

- in READ operation, the output S_i represents the i-th syndrome

  $$S_i = \oplus / (C_i, DG_i)$$

  where c_i is in MDR_{16+i} and DG_i represents 8 MDR positions as defined on the diagram. The signals S_{i0}, S_{i1} are morphic outputs for the i-th syndrome. By definition, $S_i = 1$ if there are no single errors in the positions corresponding to C_i and DG_i.

The Carter SEC/DED analyzer, shown in Figure 4-18, performs the checking of syndrome generation by morphic signal SGC, which is 1_M if there is no error in any of the syndrome generators and 0_M otherwise. This is so because odd parity is used in the encoding. Two parity trees are used to produce a morphic syndrome parity check (SPC). Since there is an even number of syndromes and parity bits and the syndrome "no error" condition is 1_M, there should be an even number of 1's in total under no error condition. Therefore, both parity trees should have like parity and SDC is 0_M under a no error condition and 1_M otherwise.

From morphic signals SGC and SPC it can be decided when there is a no syndrome error (NE), a double error (DE) or a single error (SE). These conditions are mutually exclusive and that fact can be used to provide for checking analyzer circuits as indicated by the No Circuit Error (NCE) network.

The morphic error indication signals are systematically collected in an 8-bit error status register (ESR) (see Figure 4-19). On the basis of address and data parity checking, SES/DED analyzer outputs and command/control checking, two outgoing signals, are formed. Whenever a single error has been corrected, a morphic interrupt signal SECI is generated. If an uncorrectable condition exists, the memory error interrupt (MINT) is generated. If MINT condition exists, a write operation is prevented.

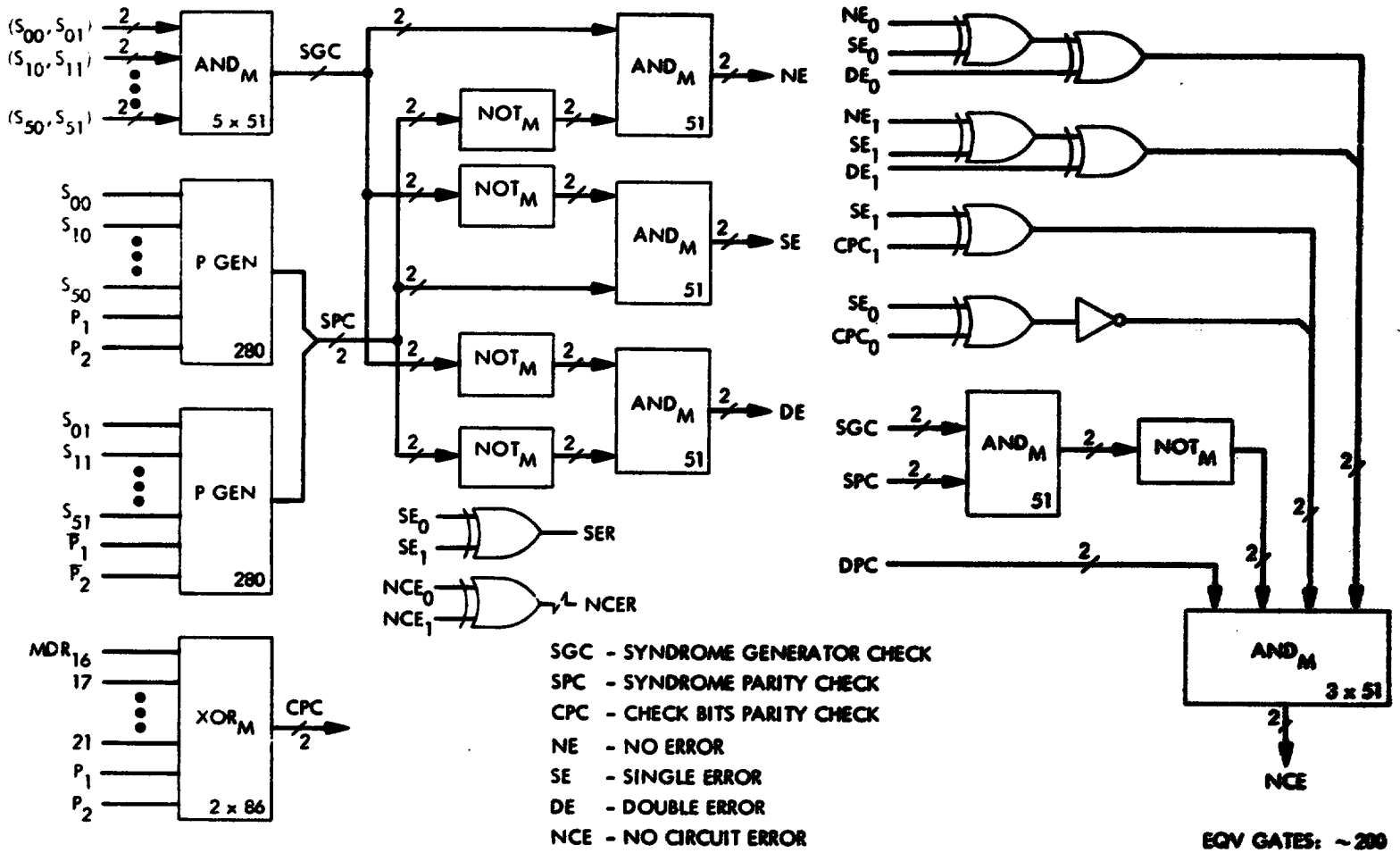


Figure 4-18. SEC/DED Analyzer (SDA)

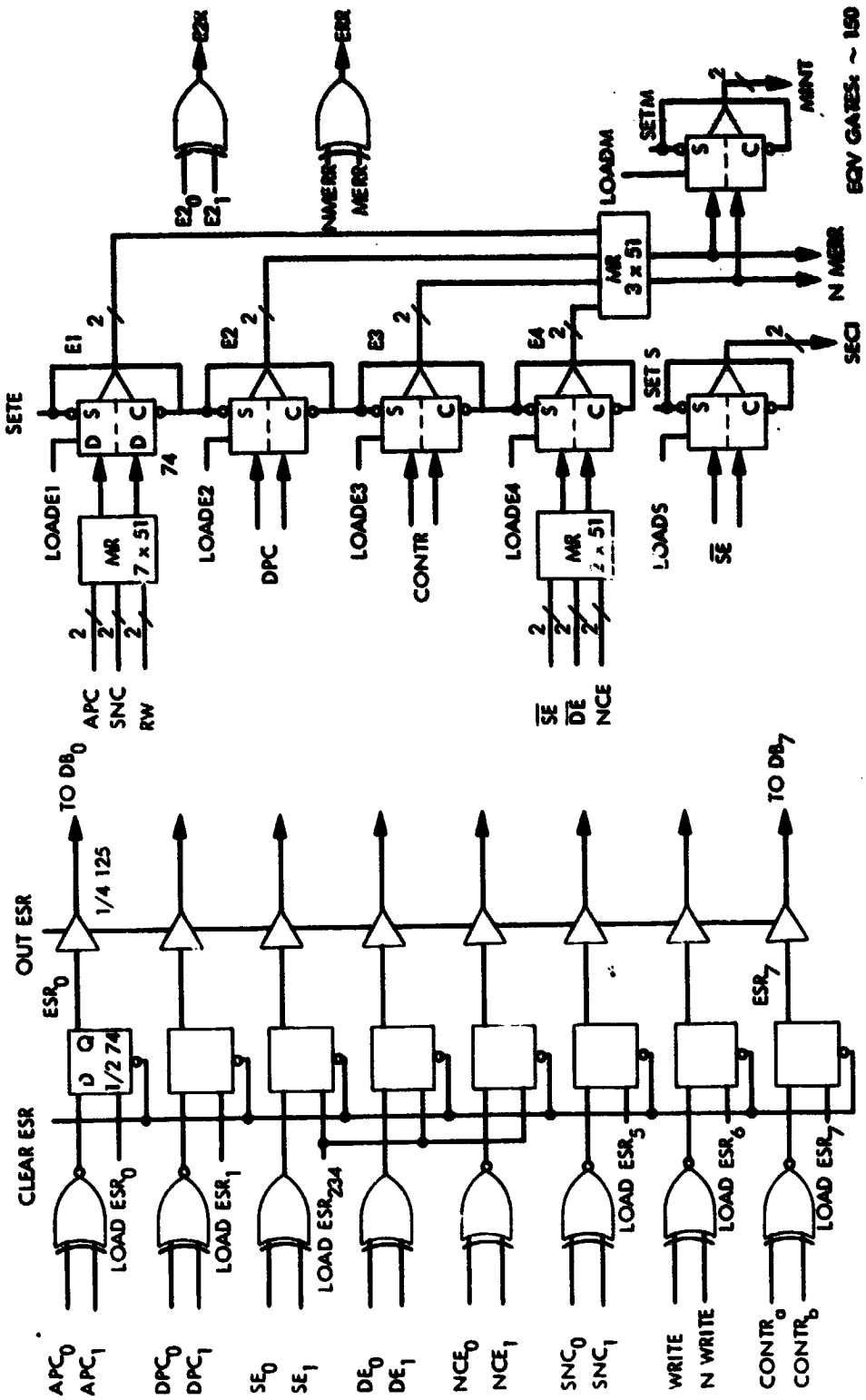


Figure 4-19. Error Status Register and Memory Interrupt (ESR/MEI)

4.1.4.4 Memory Control Section. The Memory Control Section (MCS) provides control signals required to implement operation and command algorithms. As indicated in Figure 4-20a, the MCS consists of the following subsections: the Control Interface (CI), the Clock Generator (CPG), two Condition Generators (KG_a, KG_b), two State Sequences (SS_a, SS_b), two Control Signal Generators (CSG_a, CSG_b) and a Control Signal Comparater (CSR). These subsections are described in more detail in the following paragraphs.

(a) Control Interface (CI) and Clock Generator (CPG)

The Control Interface (CI) is shown in Figure 4-20b. It consists of SCCM-MIBB handshaking circuits (MSTART-MCOMPLETE circuits), and several flags at the out-of-range command register with the command decoder. The Clock Generator, also shown in Figure 4-20b, consists of the basic 8MHz clock oscillator, a synchronizing divider which produces a 4MHz clock train in automatic mode when MSTART=1. In the manual mode, a single edge is produced. (It is assumed that all flip-flops are edge-triggered.)

(b) Condition Generator (KG)

The conditions generated by KG are defined below:

$K_1 = HAM \cdot SER \cdot SECEN \cdot NCER \cdot \overline{RTRY}$

$K_2 = HAM \cdot SER \cdot SECEN \cdot NCER \cdot RTRY \cdot \overline{RTRONE}$

$K_3 = HAM \cdot ERR \cdot \overline{SER} + \overline{HAM} \cdot E2R + RTRY \cdot RTRONE \cdot ERR$

$K_4 = RES + REA + REP + EDR + RSP + RBR + SEC$

$K_5 = ORC_a \cdot MSTART$

$K_6 = NWRITE + SSN$

$K_7 = NWRITE \cdot K_1 + SSN$

$K_8 = FORME \cdot MSTART(WRITE + NWRITE)$

$K_9 = \overline{WRITE} \cdot \overline{NWRITE} \cdot \overline{SSN} \cdot \overline{RCB}$

$K_{10} = (\overline{ERR} + K_3 + K_2)NWRITE$

$K_{11} = NWRITE + WRITE$

The implementation is straightforward and is not shown here.

(c) State Sequencer (SS)

State Sequencer (SS) implements the control state diagram shown in Figure 4-20c. The t states correspond to the steps of the operation and command algorithms given before, as follows:

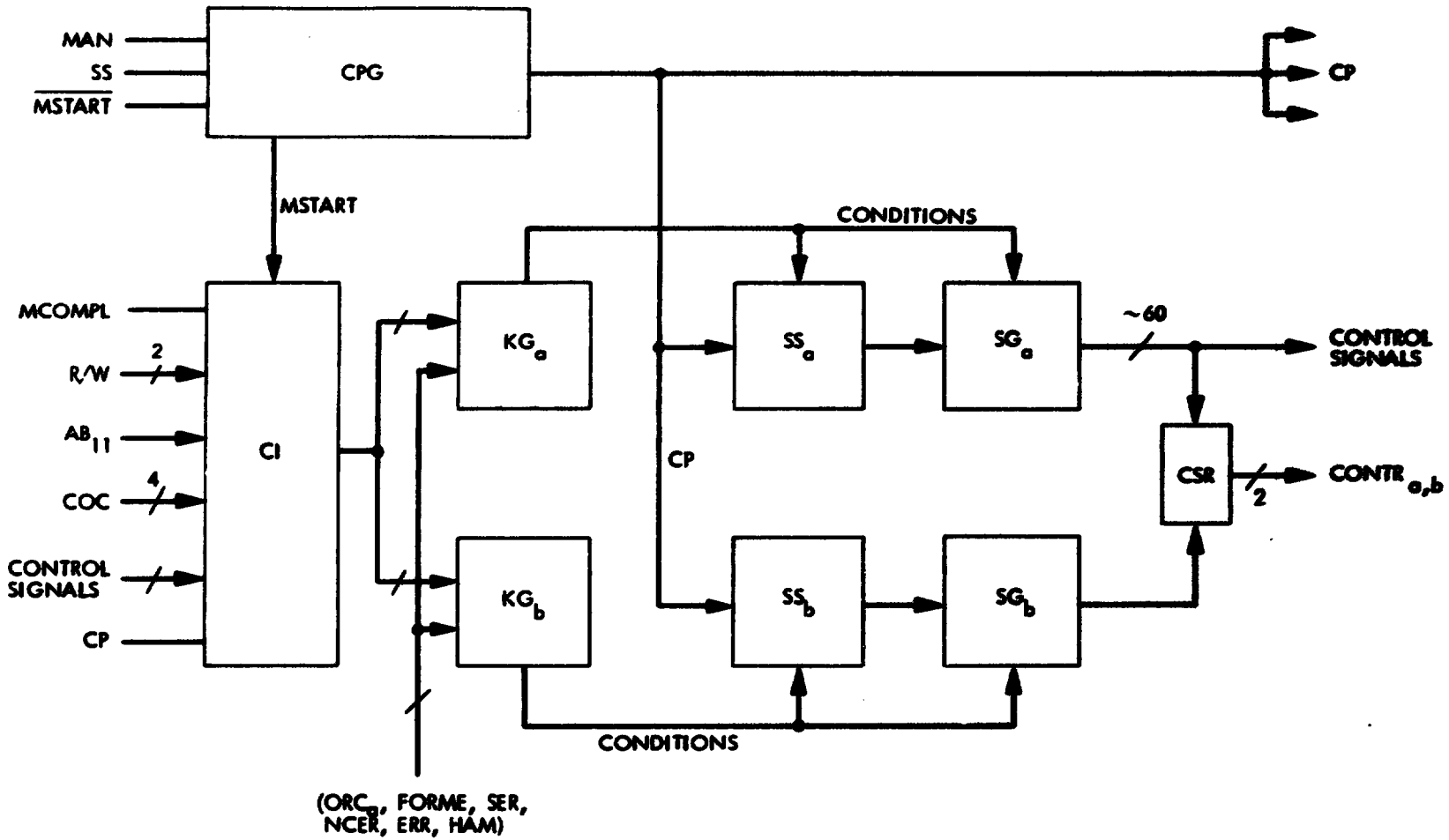


Figure 4-20a. Control Section

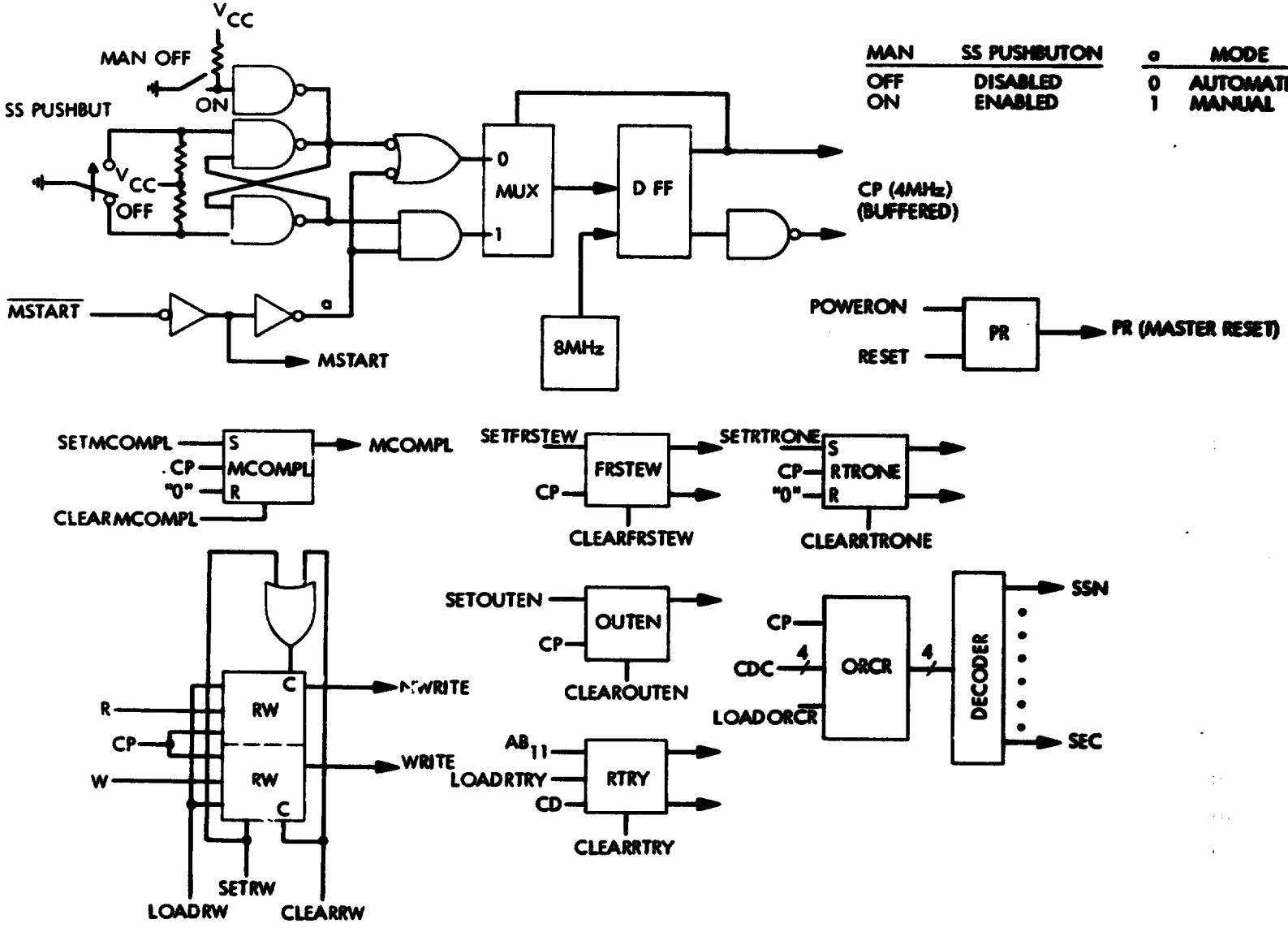


Figure 4-20b. Control Interface and Clock Generator (C1&C6)

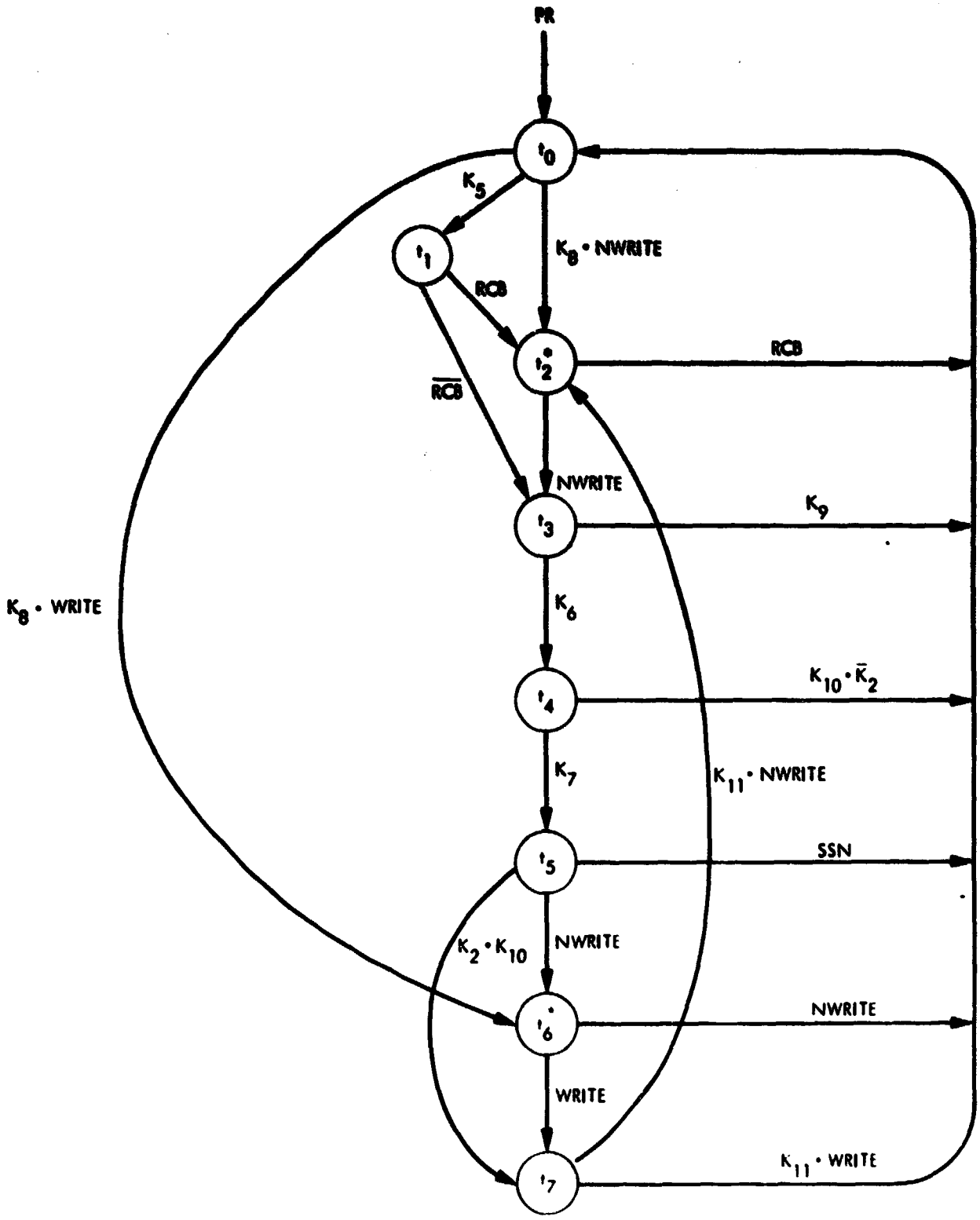


Figure 4-20c. MCS State Diagram

t_0 WAIT

t_1 DECODE

t_2* READ1, RCB1

t_3 READ2, SSN1, RES1, REA1, REP1, EDR1, RSP1, RBR1, SEC1

t_4 READ3, SSN2

t_5 READ4, SSN3

t_6* READ5, WRITE1

t_7 READ6, WRITE2

The states t_2 and t_6 take two (2) clock periods in order to accommodate the access to the storage array. The implementation for the breadboard uses a standard synchronous connector plus multiplexer approach. In a VLSI implementation it is likely that an asynchronous sequencer would be more appropriate. The state transitions of the counter T, shown in Figure 4-20d, are controlled by the following functions:

$TCOUNT = t_0 \cdot K_5 + t_1 \cdot RCB + t_2 * NWRITE + t_3 \cdot K_6 + t_4 \cdot K_7 + t_5 \cdot NWRITE + t_6 * WRITE$

$TLOAD = t_0 \cdot K_8 + t_2 * \cdot RCB + t_3 \cdot K_9 + t_4 \cdot K_{10} + t_5 \cdot SSN + t_6 * NWRITE + t_7 \cdot K_{11}$

$TCLEAR = PR + TQ_3$ (i.e. for all t_i, $i \geq 8$).

The parallel load inputs are defined as follows:

Present State	Next State I_3	I_2	I_1	I_0	Condition
t_0	0	0	1	0	NWRITE
	0	1	1	0	WRITE
t_1	0	0	1	1	$\overline{RCB}$
t_2	0	0	0	0	
t_3	0	0	0	0	
t_4	0	0	0	0	$K_3 + \overline{ERR}$
	0	1	1	1	K_2
t_5	0	0	0	0	
t_6	0	0	0	0	
t_7	0	0	1	0	NWRITE
	0	0	0	0	WRITE

Therefore, the parallel inputs are:

$I_3 = 0$

$I_2 = t_0 \cdot WRITE + t_4 \cdot K_2$

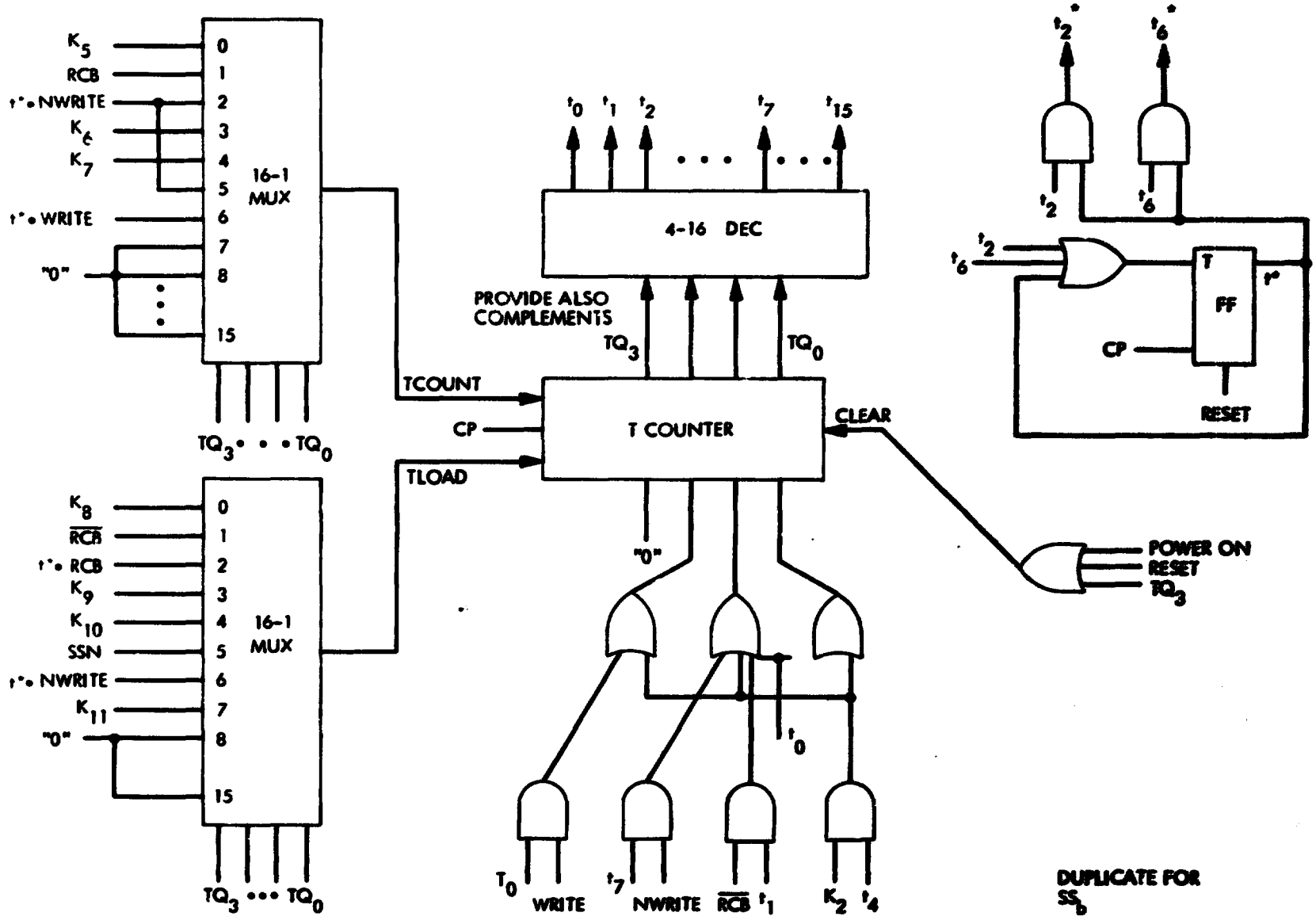


Figure 4-20d. State Sequencer (SS_a)

$I_1 = t_0 + t_1\overline{RCB} + t_4 \cdot K_2 + t_7 \cdot NWRITE$

$I_0 = t_1\overline{RCB} + t_4 K_2$

The sequencer is shown in Figure 4-20d. For additional flexibility in the breadboarding phase, we use Mod-16 counter and 16-to-1 multiplexers even though Mod-8 counter and 8-to-1 multiplexers would be sufficient.

(d) Control Signal Generator (CSG)

The control signals for register and selection networks are defined below. Again, the implementation is simple and it is not shown here. Since there is a large number of control signals (approx. 60), direct morphic reduction would be too costly. However, it is possible to group together (by ORing) mutually exclusive signals before reduction. For bread-boarding phase, a direct signal-to-signal comparison on equivalence is preferred. A control error is indicated if not all comparisons are the same.

EBAR register

SETEB=PR+$t_7 \cdot$NWRITE+$\overline{\text{MSTART}} \cdot$MCOMPL

LOADEB=$t_5 \cdot$NWRITE

EWAR register

LOADEW=(t_2*$\cdot$NWRITE+t_6*$\cdot$WRITE)$\overline{\text{FRSTEW}}$

FRSTEW flag

CLEARFRSTEW=PR+$t_3 \cdot$REA

SETFRSTEW=$t_4 \cdot$NWRITE$\cdot K_3 + t_7 \cdot$WRITE$\cdot$ERR

E register (morphic)

SETE=PR+$t_7 \cdot$NWRITE

LOADE1=t_2*$\cdot$NWRITE+t_6*$\cdot$WRITE

LOADE2=$t_3 \cdot$NWRITE+t_6*$\cdot$WRITE

LOADE3=t*+t'$_2$t'$_6$

LOADE4=$t_3 \cdot$NWRITE

ESR — register

CLEARESR = $PR+t_7 \cdot NWRITE+\overline{MSTART} \cdot MCOMPL$

$LOADESR_0 = t_2{}^* \cdot NWRITE+t_6{}^* \cdot WRITE$

$LOADESR_1 = t_3 \cdot NWRITE+t_6{}^* WRITE$

$LOADESR_2 = t_3 \cdot NWRITE+t_7 \cdot WRITE$

$LOADESR_3 = LOADESR_2$

$LOADESR_4 = LOADESR_2$

$LOADESR_5 = LOADESR_0+t_5 \cdot SSN$

$LOADESR_6 = LOADESR_0$

$LOADESR_7 = t^*+t'_2 \cdot t'_6$

MCOMPL — flag

SETMCOMPL = $PR+t_4 \cdot NWRITE \cdot K_3+t_6{}^* \cdot NWRITE+t_7 \cdot WRITE+t_5 \cdot SSN+t_3 \cdot K_4+t_2{}^* \cdot RCB$

CLEARCOMPL = $\uparrow MSTART(FORME+ORC_a)$

MAR — register

LOADMAR = $t_0(FORME+ORC_a \cdot RCB) \uparrow MSTART$

MDR — register

LOADMDR = $t_0 \cdot FORME \cdot MSTART \cdot WRITE+(t_2{}^*+t_5+t_4 \cdot K_2) \cdot NWRITE$

MCR — register

LOADMCR = $t_6{}^* \cdot WRITE \cdot HAM+t_2{}^* \cdot RCB$

OUTEN — flag

CLEAROUTEN = $PR+\overline{MSTART} \cdot MCOMPL$

SETOUTEN = $t_3(NWRITE+K_4)+t_2{}^*RCB$

RTRONE — flag

CLEARRTRONE = $PR+\overline{MSTART} \cdot MCOMPL$

SETRTRONE = $t_7 \cdot NWRITE$

RTRY	flag	
	CLEARRTRY =	PR
	LOADRTRY =	$t_3 \cdot EDR$
ORCR	register	
	LOADORCR =	$t_0 \cdot ORC_a \cdot MSTART$
POSAR	$SETPOS_a$ =	$PR+t_3 \cdot RBR \cdot AB_{11}$
	$SETPOS_b$ =	$PR+t_3 \cdot RBR \cdot \overline{AB}_{11}$
	LOADA =	$t_3 \cdot RSP \cdot AB_{11}$
	LOADB =	$t_3 \cdot RSP \cdot \overline{AB}_{11}$
P	register	
	LOADP =	$t_0 \cdot FORME \cdot MSTART \cdot WRITE+(t_2^*+t_6^*) \cdot NWRITE$
RW	register	
	SETRW =	$t_7 \cdot NWRITE$
	LOADRW =	$t_0 \cdot FORME \cdot MSTART$
	CLEARRW =	$\overline{MSTART} \cdot MCOMPL$
	RESETRW =	$t_4 \cdot K_2$
SECI	register	
	SETS =	$PR+t_7 \cdot NWRITE+\overline{MSTART} \cdot MCOMPL$
	LOADS =	$t_5 \cdot NWRITE$
MINT	register	
	SETM =	$PR+t_7+\overline{MSTART} \cdot MCOMPL$
	LOADM =	$t_4 \cdot K_3+t_7+t_5 \cdot SSN$
SECEN	flag	
	SETSECEN =	PR
	LOADSECEN =	$t_3 \cdot SEC$
SNR	register	
	$SETSNR_6$ =	$t_3 \cdot SSN$
	LOADSNR =	$t_4 \cdot SSN$

DBOUT Signals

$OUTRA = OUTEN \cdot REA \cdot MCOMPL$

$OUTESR = OUTEN \cdot RES \cdot MCOMPL$

$OUTMCR = OUTEN \cdot RCB \cdot MCOMPL$

$OUTMDR = OUTEN \cdot MWRITE \cdot MCOMPL$

$OUTEBA = OUTEN \cdot REP \cdot MCOMPL$

SELDW Selection

$SELDW = t_2^* \cdot MWRITE$

MEMEN (Memory Enable)

$MEMEN = \overline{ERR}$

ERR

$ERR = \overline{NMERR_0 \oplus NMERR}$

$SER = SE_0 \oplus SE_1$

$NCER = \overline{NCE_0 \oplus NCE_1}$

4.1.5 Estimated Complexity of Implementation

The design of the MIBB was directed toward partitioning into LSI modules of 500-750 gates per module. This has been largely achieved, as summarized in Table 4-2. It is also small enough to be implemented on a single VLSI circuit.

The breadboard realization using SSI/MSI modules requires about 200 chips.

Table 4-2. Component Count

Module	Equivalent Gates	I/O Pins	LSI Chips
ABI	∿ 465	· 64	1
ECS	∿ 650	·100	1
DBSA	∿ 775	· 64	1
MCS	∿ 500	· 64	1
			(Duplicate)
	∿3000		5 + 2

4.2 THE CORE BUILDING BLOCK

The Core Building Block (Core-BB) is responsible for (1) detecting CPU and bus faults, (2) collecting fault indications from the other building blocks, and (3) disabling its computer module upon the detection of a permanent fault. Two fault-handling options are provided by the Core-BB:

(1) Stop at the first fault indication;

(2) Rollback at first fault indication, stop if fault recurs

4.2.1 Core Building Block Requirements

Specific requirements of the Core Building Block are listed below:

(1) Compare two CPU's for disagreement;

(2) Parity encode CPU output for internal bus transmission;

(3) Check parity on internal bus;

(4) Recognize Core-BB commands: Halt and Inhibit, Restart, and Enable, as out-of-range addresses;

(5) Allocate the internal bus amongst several DMA modules;

(6) Detect internal faults within the Core-BB;

(7) Collect internal fault indications from all building blocks within the computer module;

(8) Disable SCCM output (or set error message) under fault conditions;

(9) Provide reset/halt, or reset/rollback, capability for optional transient fault recovery;

(10) Halt computation on recurring faults.

A block diagram of the Core-BB is shown in Figure 4-21.

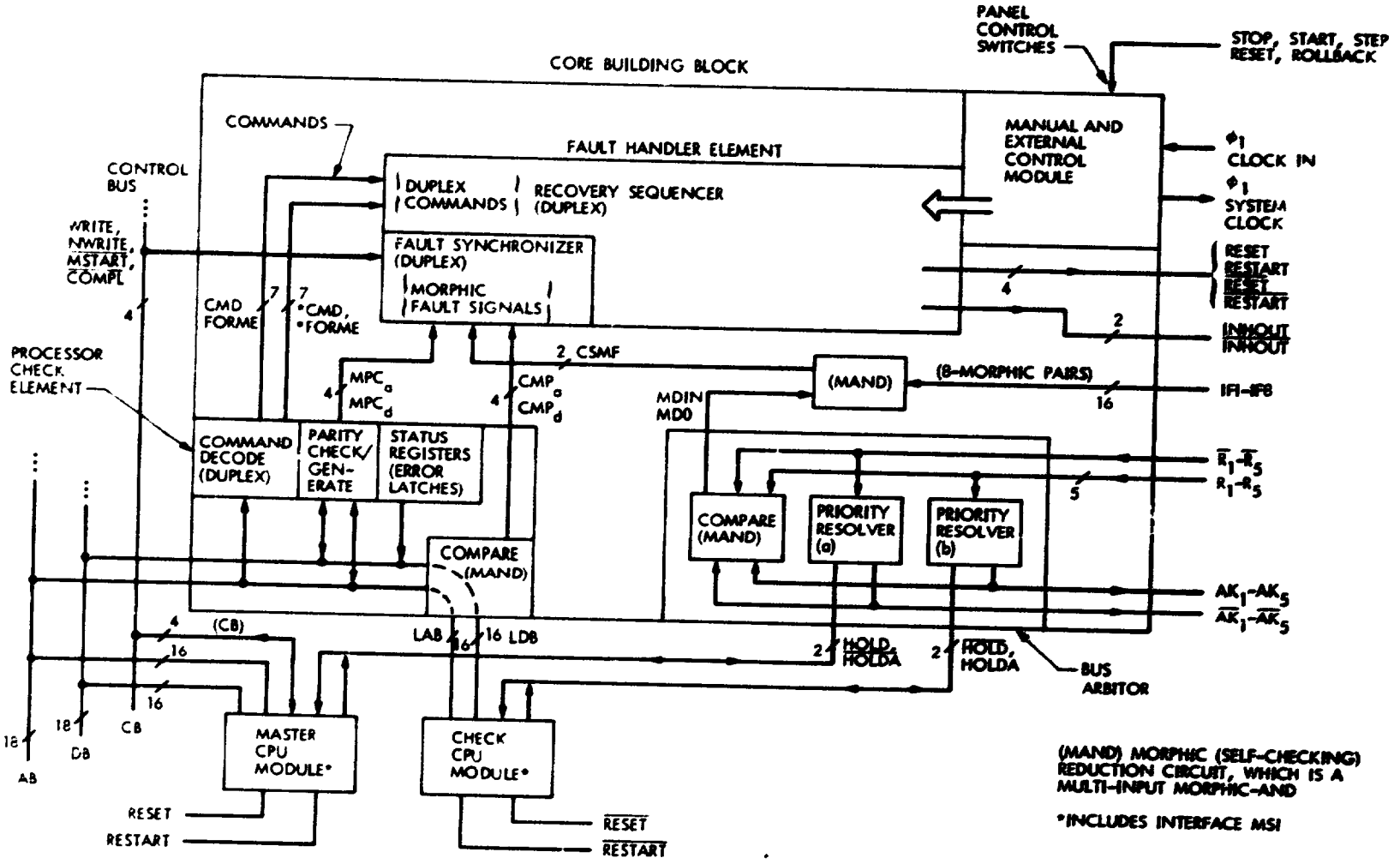


Figure 4-21. Core-BB Block Diagram

4.2.1.1 Core Building Block Connections. The following is a listing of Core-BB connections and a brief description of their function:

(1) Internal Data Bus (DB) 18 lines (16 + 2 parity)

(2) Internal Address Bus (AB) 18 lines (16 + 2 parity) - All building blocks and the Master CPU are connected to these buses which tie together to the SCCM. The Core-BB checks parity on outputs from other modules and generates parity when the Master CPU outputs on either bus.

(3) Local Data Bus (LDB) 16 lines - Special data bus to the check CPU. The Core-BB passes data directly from the Internal Data Bus to the Local Data Bus when inputs are required by both CPU's. When both CPU's output, the Core-BB compares the two processors by comparing the two data buses.

(4) Local Address Bus (LAB) 16 lines - Carries address outputs by the Check CPU which are compared with address outputs of the Master CPU (by comparing the Internal Address Bus with Local Address Bus).

(5) R1-R5 (5) - Bus Request signals from DMA controllers in other building blocks.

$\overline{R1}$-$\overline{R5}$ (5) Complement of R1-R5

These signals form morphic pairs (R_1, $\overline{R}_1$, ..., R_5, $\overline{R}_5$), which are sent from up to five SCCM building blocks. They are checked for proper coding (i.e., being complementary). The true values (R1-R5) and the complement requests ($\overline{R}_1$-$\overline{R}_5$) are processed by two redundant circuits within the Core-BB, which in turn generate a true and complementary set of acknowledge signals (Ak1-A5). ($\overline{Ak1}$-$\overline{Ak5}$).

(6) Ak1-Ak5 (5) - Bus Grant (Acknowledge) signals to DMA channels. $\overline{Ak1}$-$\overline{Ak5}$ (5) - complement of Ak1-Ak5 forming morphic pairs.

(7) HOLD $\overline{HOLD}$ (2) - Bus Release request to Master and Check CPU's.

(8) HOLDA, $\overline{HOLDA}$ (2) - Bus Release acknowledge from Master and Check CPU's.

(9) IF1-IF8, $\overline{IF1}$-$\overline{IF8}$ (16) - Eight morphic Internal Fault indicators from other building block circuits.

(10) RESET, $\overline{RESET}$ (2) - Morphic reset signals to all SCCM modules from duplicated logic in the Core-BB.

(11) INHOUT, $\overline{INHOUT}$ (2) - Inhibit outputs to Bus Controller and I/O BB's, from duplicated logic in Core-BB.

(12) RESTART, $\overline{RESTART}$ (2) - Morphic restart signals to Master and Check CPU from duplicated logic in Core-BB.

(13) ϕ_1 Clock In (1) - 1 Mhz square wave clock in to Core-BB.

(14) ϕ_1 (1) System Clock - Clock to all circuitry in SCCM, sent from and controlled by Core-BB.

(15) WRITE, NWRITE (2) - Memory read/write control level of the SCCM Internal Bus.

(16) $\overline{MSTART}$ (1) - Memory Start Signal of SCCM Internal Bus.

(17) $\overline{COMPL}$ (1) - Completion level of SCCM Internal Bus.

Counting Vcc and ground, this circuit will require a 128 pin package.

4.2.2 Core Building Block Implementation

The Core Building Block consists of three sub-elements: A Processor Check Element, A Bus Arbitration Element, and a Fault Handler Element, which are described below.

4.2.2.1 The Processor Check Element (PCE). The Processor Check Element serves three functions: (1) to compare the outputs of two synchronous processors; (2) to encode and check internal bus parity; and (3) to recognize and decode commands sent to the Core through the internal SCCM bus.

The PCE is shown in Figure 4-22. It is connected to the two 18-bit internal address and data buses within the SCCM. The Master CPU and other building blocks in the SCCM also reside on these buses. The PCE provides a local address and data bus for a Check CPU, which is operated synchronously with the Master CPU, and its outputs are compared for checking.

Internal circuits in the PCE consist of:

(a) Morphic Comparators MCMP - Each of these circuits compares two pairs of 16-bit inputs, and generates a two-wire output. The output takes on values 0,1 or 1,0 if the 16-bit inputs agree, and they take on values 1,1 or 0,0 if the inputs disagree or if an internal fault occurs in the comparator circuit. These circuits are said to be self-checking in that nearly all internal faults will eventually result in an error indication.

One MCMP circuit compares the address output of the two processors. The second compares their outputs to the data bus. An isolation circuit is provided so that input data to the Master CPU can also be passed to the check CPU.

(b) Morphic Parity Check/Generator - Two circuits are provided to check and generate parity on the address and data buses respectively. Coding on each bus consists of two odd parity bits; one over all even bits and one over all odd bits. Since the Master CPU generates 16 bit address and data outputs without parity, the parity generators add the extra two parity bits to

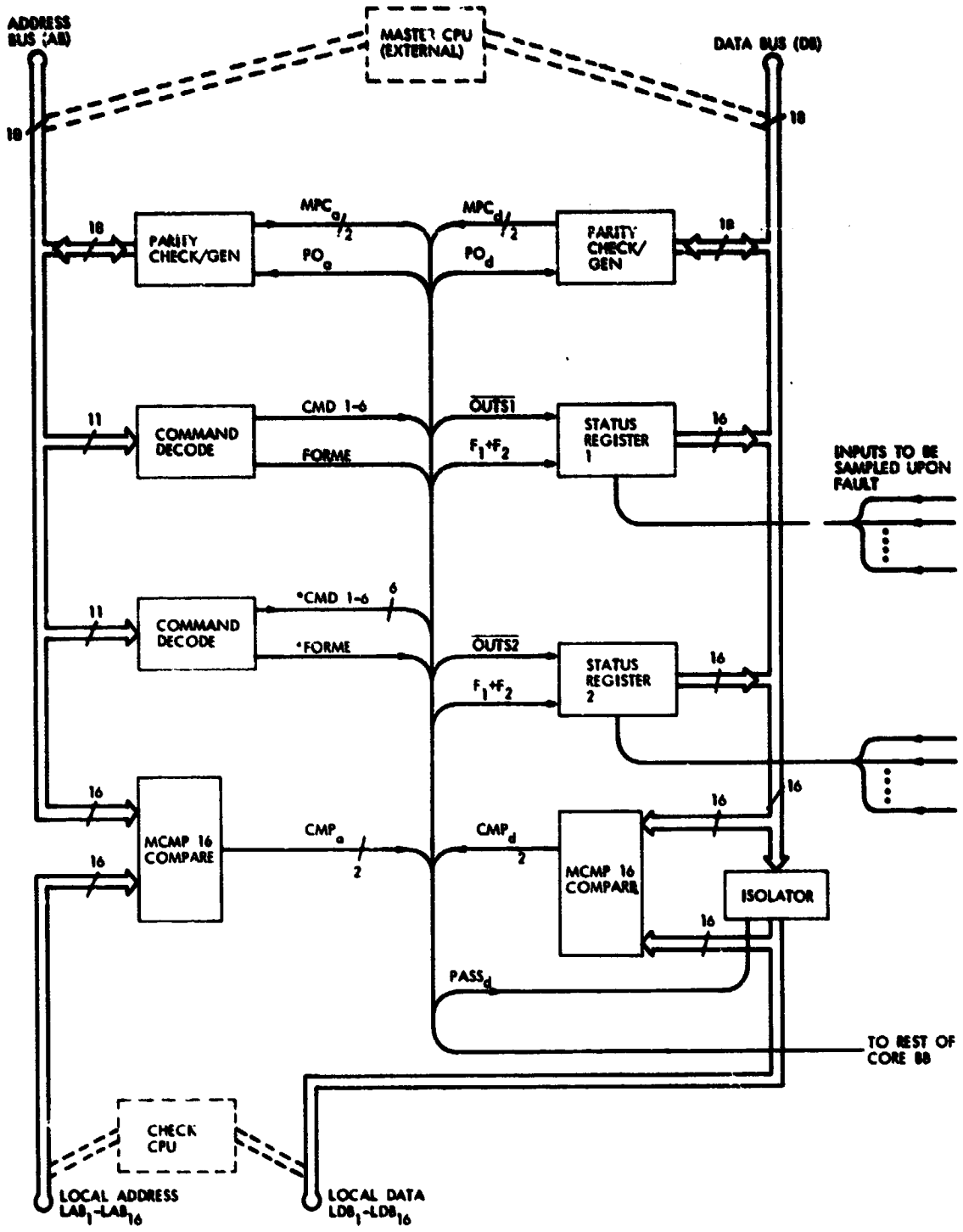


Figure 4-22. The Processor Check Element

their associated bus during CPU outputs. Other data on the internal buses is expected to be coded (e.g., memory data) and the parity checkers check for proper coding.

Each self-checking (morphic) parity check generates a two-wire output with values 1,0 and 0,1, which represent a correct check, and 1,1 and 0,0, which represent either uncoded data on the associated bus or a fault in the checker.

(c) Command Decoders. Two command decoders are provided which have identical outputs. When an out-of-range address appears on the address bus (with AB_0-AB_3 = 1111) and the Core-BB is addressed (AB_4-AB_7 = 0001), the three least significant bits of the address bus are decoded to generate six commands. These are designated CMD1-CMD6 (*CMD1-*CMD6 from the duplicate decoder). If any of these commands are received, the level FORME is raised. The outputs of the command decoder are compared in the Fault Handler to detect faults in this circuitry.

Core-BB Commands are:

CMD1 - START Clock
CMD2 - STOP Clock
CMD3 - Initiate Rollback
CMD4 - Clear Faults, Enable Outputs
CMD5 - Output Error Status Word 1
CMD6 - Output Error Status Word 2

(d) Status Registers. Two status registers are used to sample various fault indicators and make this information available to external computer modules. When a fault is detected (F_1+F_2) by a fault synchronizer, this data is sampled (i.e., clocked into the status registers). Two Core-BB commands are reserved to read out the status registers. When the level $\overline{OUTS}$ goes

low, tri-state drivers are enabled in the respective status register and its data is output to the data bus.

Figure 4-23 shows a preliminary logic design of the circuits which make up the Processor Check Element. Specific interface signals are:

Input to DCE:

PO_a-PO_d	- Generate and output parity on address or data bus, respectively.
$\overline{OUTS1}$	- Output Status Register 1 to SCCM data bus.
$\overline{OUTS2}$	- Output Status Register 2 to SCCM data bus.
F_1+F_2	- Load status registers (a fault is detected).
$PASS_d$	- Connect Local Data Bus with Internal SCCM data bus.

Outputs from DCE

MPC_a, MPC_d	- Two-wire morphic parity check results for address and data bus, respectively.
CMP_a, CMP_b	- Two-wire morphic comparison results for address and data bus, respectively.
CMD1-6	- Command lines for decoded commands to the Core-BB.
FORME	- Indicates Core-BB has been commanded by an out of range address.
*CMD1-6, *FORME	- Duplicate of command decoder signals above.

4.2.2.2 The Bus Arbitor Element. The bus arbitor accepts internal bus requests ($R,\bar{R}$) from up to five DMA channels in other building block circuits. It accepts multiple requests on the basis of priority, requests release of the internal bus by the two processors (HOLD), and upon compliance by the processors (HOLDA) it grants access to the selected DMA controller (Ak). Incoming bus requests are morphic signal pairs which take on values of 0,1 when access is requested and values 1,0 when no

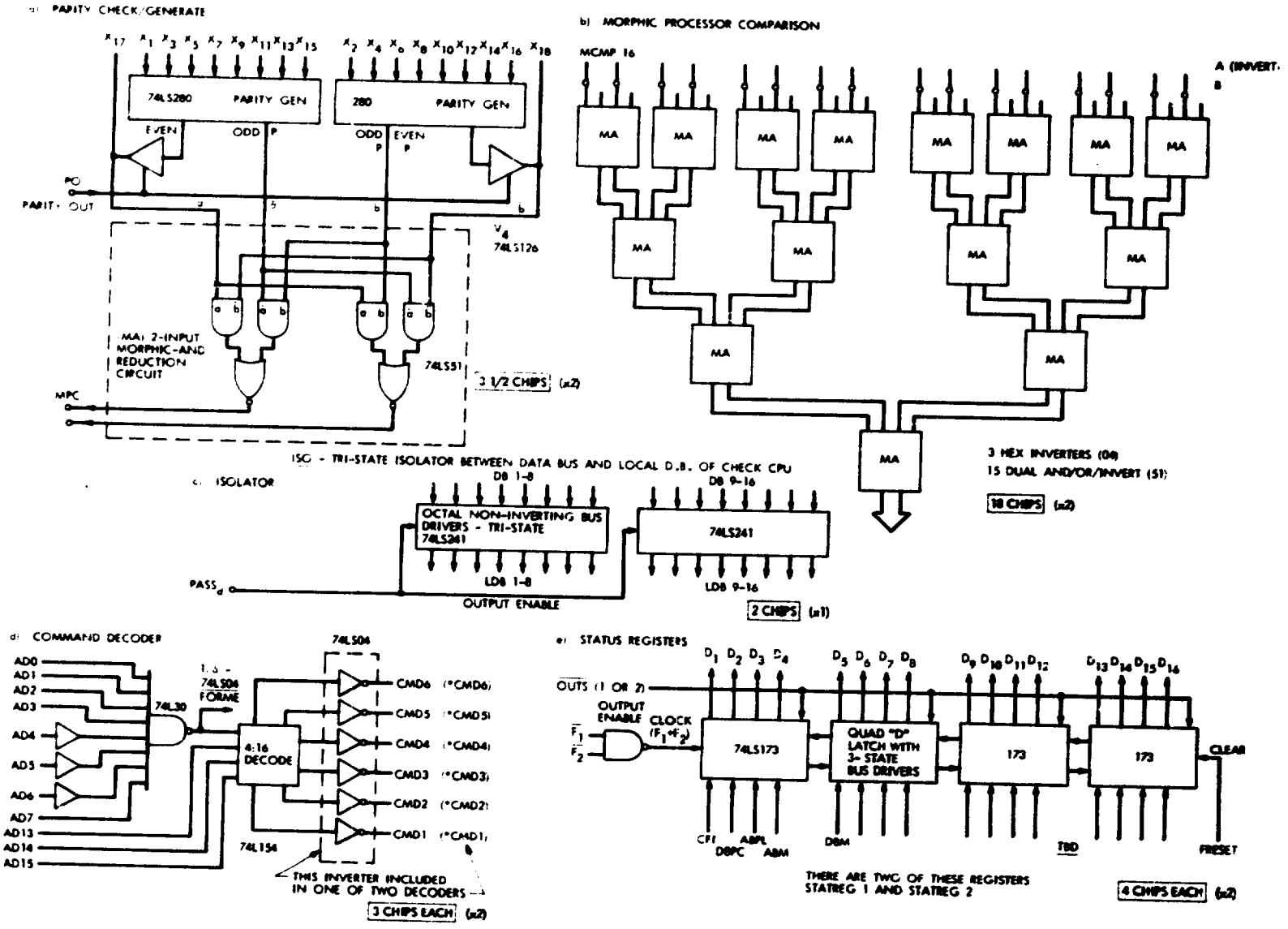


Figure 4-23. Processor Check Element Logic: (a) Parity Check/Generate; (b) Morphic Processor Comparison; (c) Isolator; (d) Command Decoder; (e) Status Registers.

request is made. Values 1,1 and 0,0 represent fault conditions. Similarly, the acknowledge signals are morphic with 01, and 1,0 representing not-acknowledge and acknowledge (grant). As before, the values 00, and 11 represent fault conditions. One variable of each morphic pair is associated with one priority resolver circuit and the other is associated with the second resolver.

The two priority resolvers are duplicated circuits, each of which provide the bus arbitration function — one in "true" logic and the other in "complement" logic. They are compared using morphic-and circuits to detect faults in either unit by their disagreement. Each priority resolver is a simple sequential circuit which accepts bus request inputs, obtains release of the internal bus by the CPU, and grants bus access to the requesting DMA channel with highest priority. A functional block diagram of the Bus Arbitor Element is shown in Figure 4-24, and a logic description of the Priority Resolver is presented in Figure 4-25. The morphic-and circuits are shown in Figure 4-26. Synchronization of the two priority resolvers is described below:

(a) Timing. A square wave clock is sent to all building blocks in the SCCM and is used for synchronization. In the first half of the cycle, the clock is high and this signal is designated ϕ_1. The inverse of the clock is ϕ_2. Thus, the rise of ϕ_1 is the beginning of a clock cycle and the rise of ϕ_2 is the middle.

All bus request, interrupt, and fault indicators are constrained to change only at the beginning of ϕ_1 and it is assumed that they are generated by a D or J/k flip-flop clocked with ϕ_1. If these incoming signals are examined instantaneously at the rise of either ϕ_2 or ϕ_1, they are assumed stable. Transmission delays prevent change at the rise of ϕ_1 and the circuits have settled by the rise of ϕ_2.

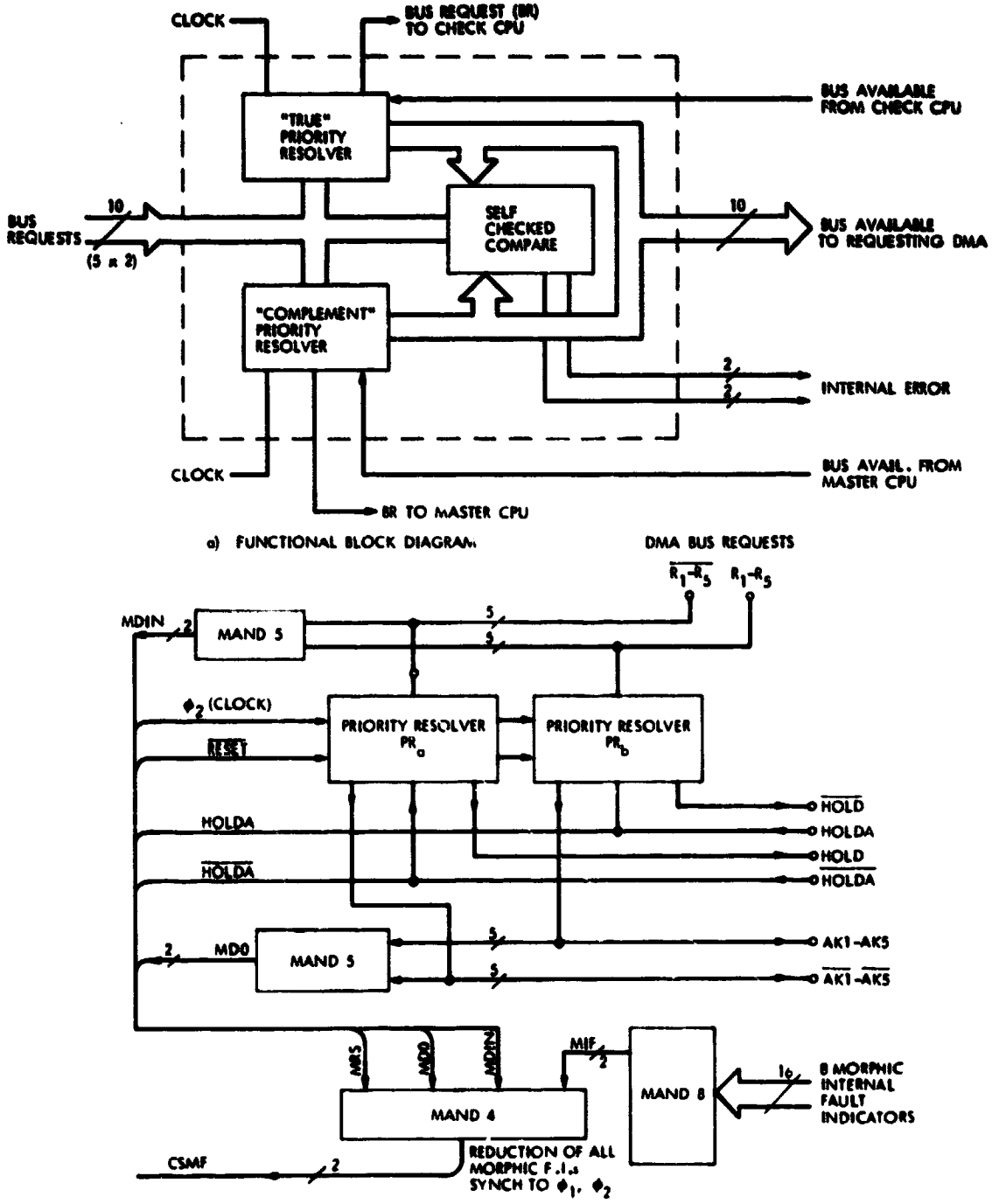


Figure 4-24. Bus Arbitor Layout

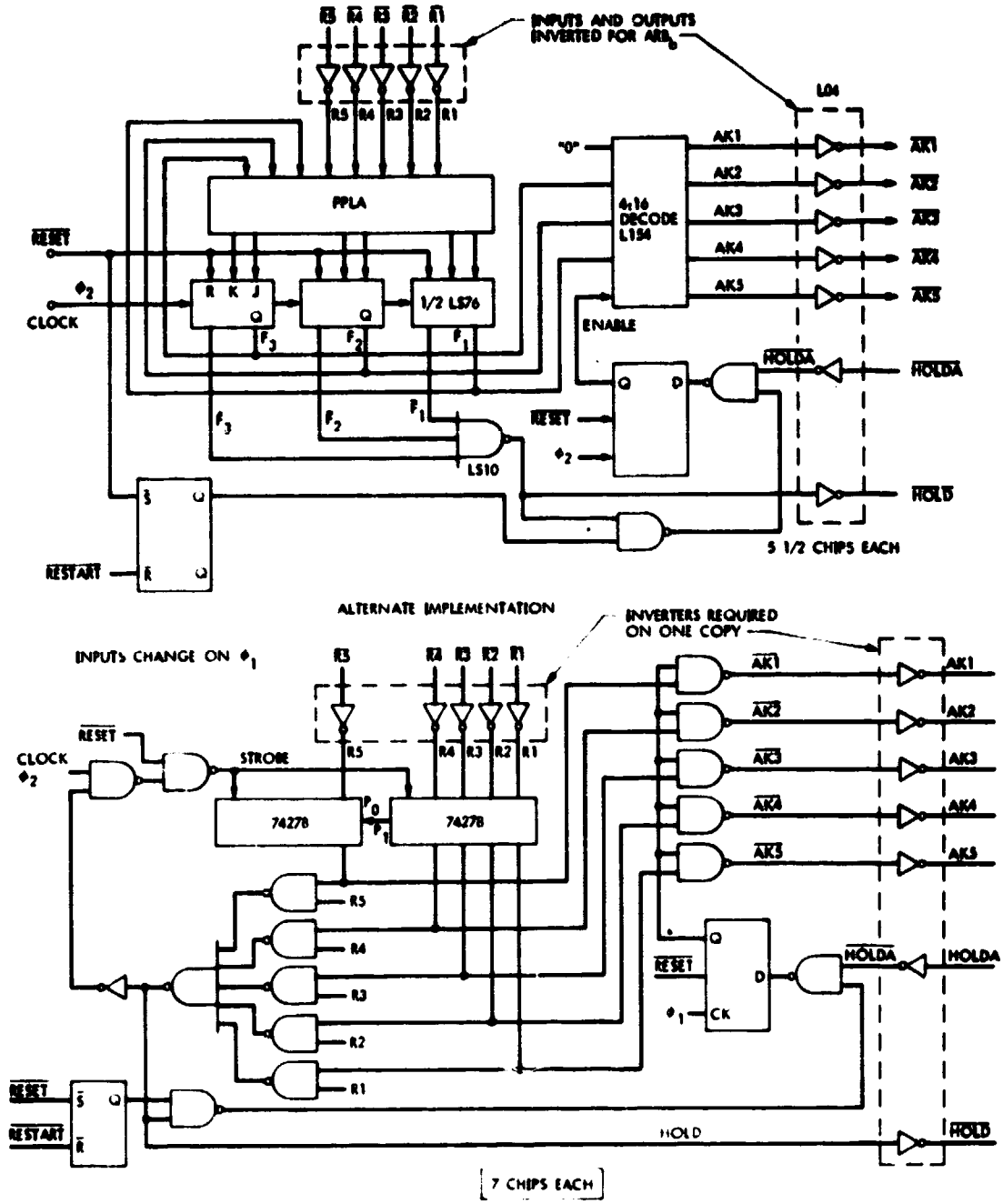
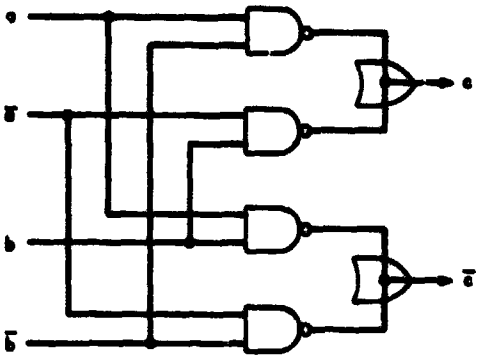


Figure 4-25. Priority Resolver Logic

A CORRECT MORPHIC PAIR TAKES ON VALUES 01 OR 10. A FAULTY MORPHIC PAIR TAKES ON VALUES 00, OR 11. THE FOLLOWING SELF-CHECKING (MORPHIC-AND) CIRCUIT REDUCES TWO INPUT MORPHIC PAIRS TO A SINGLE OUTPUT MORPHIC PAIR



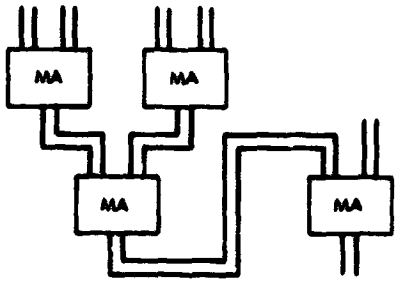
CODED INPUTS				OUTPUT	
a	ā	b	b̄	c	c̄
0	1	0	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	0	1	0	1	0

UNCODED INPUTS				OUTPUT	
0	0	X	X	1	1
X	X	0	0	1	1
1	1	0	X	0	0
1	1	X	1	0	0
1	X	1	1	0	0
X	1	1	1	0	0

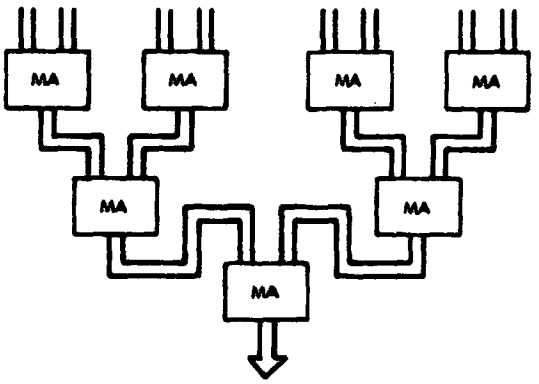
AN EXAMINATION OF THIS CIRCUIT WILL SHOW THAT FOR EVERY "STUCK" VARIABLE, THERE EXISTS AT LEAST ONE CODED INPUT WHICH WILL RESULT IN A FAULT INDICATION AT THE OUTPUT ac = 00 OR 11. THIS IS THE SELF-CHECKING PROPERTY. A TREE OF THESE CIRCUITS CAN BE USED TO COMPARE A LARGER NUMBER OF SIGNALS

a) SELF-CHECKING EXCLUSIVE-OR REDUCTION CIRCUIT OR TWO INPUT MORPHIC AND

MULTIPLE INPUT MORPHIC-AND (MAND) CIRCUITS CONSIST OF TREES OF THE CIRCUIT IN a ABOVE AND REDUCE MULTIPLE MORPHIC PAIRS TO A SINGLE TWO-WIRE PAIR TAKING ON 01 OR 10 IF ALL INPUT PAIRS ARE COMPLEMENTARY, AND 11 OR 00 IF ALL INPUTS ARE NOT COMPLEMENTARY OR A CIRCUIT ERROR IS UNCOVERED



5 INPUT MORPHIC AND



8 INPUT MORPHIC AND

b) REDUCTION TREES

Figure 4-26. Morphic and Currents: (a) Self-Checking, Exclusive, or Reduction Circuit; (b) Reduction Trees

Incoming bus request pairs and outgoing acknowledge pairs are reduced with 5-pair morphic AND circuits to generate two 2-wire morphic check signals. MDIN verifies that the inputs are correct and MDO checks that the duplex arbitors agree. These check signals are examined precisely on the rise of ϕ_1 in the Fault Handling Logic.

(b) Implementation. Two priority resolver implementations are given; one with a PLA and the other is built around a priority encoder chip (74278). The PLA is better for VLSI layout, and the other approach is easier for breadboarding (see Figure 4-25).

The resolver implementation is straightforward and the logic is largely self-explanatory. One additional feature is an added flip-flop which has a subtle but important purpose. When the system is RESET upon error, the CPU will not necessarily release itself from the bus. Thus we force isolation of the processor with tri-state transceivers and must also generate a hold acknowledge HOLDA signal. Upon detecting a permanent fault, a latch is set in the resolver which generates a continuous HOLDA signal. It is only released upon a command to restart the CPUs in a program rollback (RESTART).

4.2.2.3 Combining Fault Indicators and Other Synchronized Morphic Check Signals. There can be up to eight morphic internal fault indicators from external building blocks. These signed pairs make transitions between values 0,1 and 1,0 if their associated building block is working properly. Values 00 or 11 indicate an internal fault.

These signals are reduced by an 8-pair morphic-and circuit to produce a single morphic internal fault indicator MIF. Since the internal fault and bus arbitor check signals are all synchronized with the ϕ_1 clock, they can be combined into a single 2-wire morphic fault indicator. Thus, MIF, MDIN, and MDO are combined with a 4-input

morphic-and circuit to produce a single CSMF fault indicator which is a combination of all Clock-Synchronized Morphic Fault indicators. (An additional synchronous morphic indicator MRS is included which is the result of comparing the outputs of two duplex Recovery Sequencers in the Fault Handling Logic.)

4.2.2.4 The Fault Handler Element. The Fault Handler Element is responsible for overall fault detection in the SCCM and is also capable of taking limited recovery action. It consists of two major parts; duplex Fault Synchronizers, and duplex Recovery Sequencers. Both parts are duplicated and compared to provide fault detection, as shown in Figure 4-27.

Each Fault Synchronizer examines morphic fault indicators and check signals from the other building blocks and from within the Core-BB itself. Its primary function is to examine these signals only when they are stable and valid to detect faults, and to deliver a Master Fault Indicator to the Recovery Sequencer pair.

The Recovery Sequencer (upon receiving a Master Fault Indicator), disables outputs from the SCCM and resets the CPU's. Optionally, a restart can be attempted, and if successful, the software can re-enable outputs and clear the fault indications in the Fault Handler.

Either of the Fault Synchronizer-Recovery Sequencer pairs can disable outputs from the SCCM. Also Recovery Sequencer outputs are compared and a disagreement is signalled to both Fault Synchronizers.

A logic diagram for a Fault Synchronizer and Recovery Sequencer is given in Figure 4-28 and is described below:

(a) The Fault Synchronizer. This circuit examines the various morphic fault indicators (CMP, MPC, CSMF) at times when (1) their checks are relevant, and (2) when the morphic signals are stable (not changing). The clock-synchronized morphic fault indicator CSMF is examined at the rise of every ϕ_1. The comparison and parity check pairs are examined when a bus completion signal (COMPL) is observed. The five morphic check

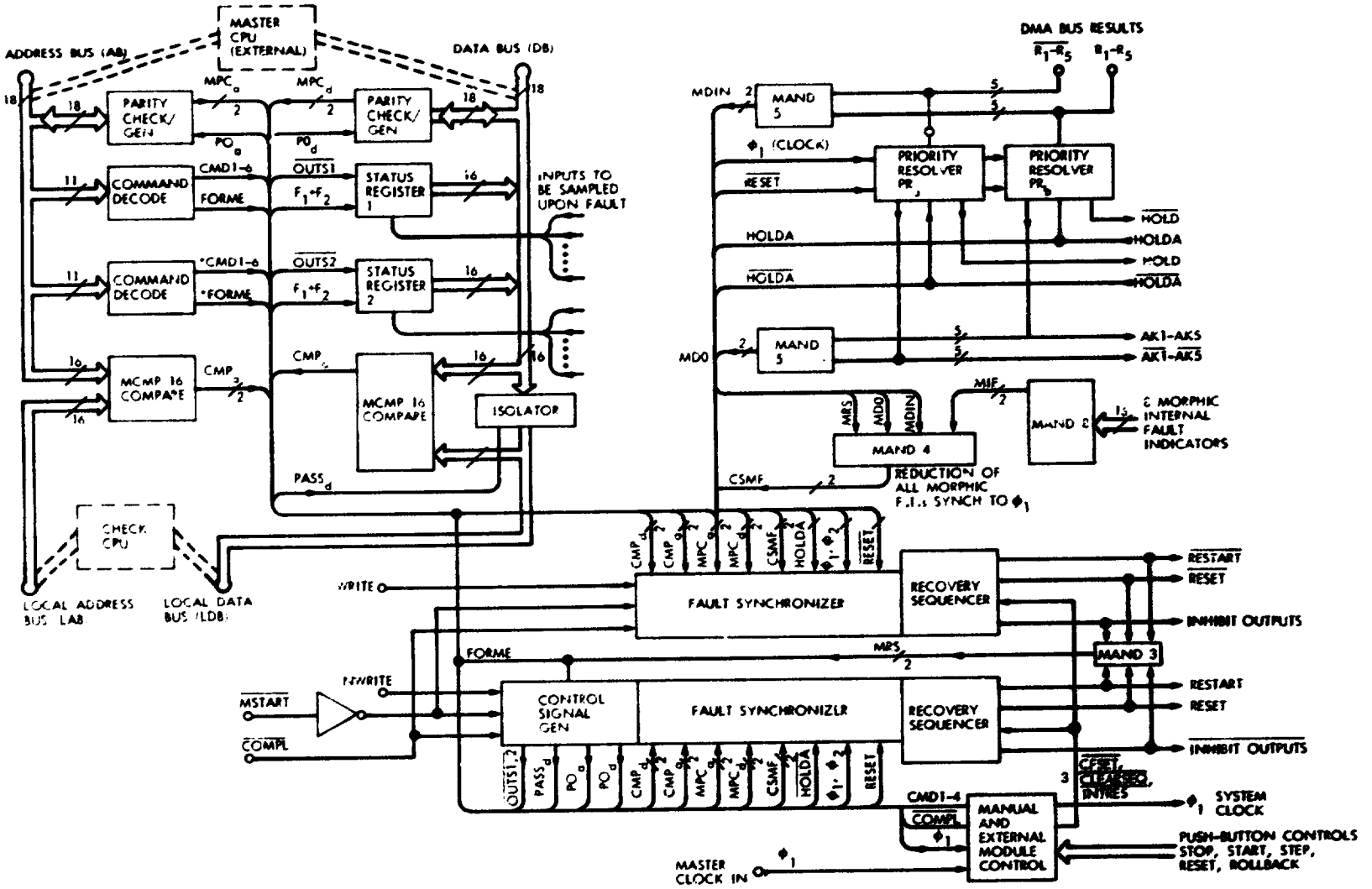


Figure 4-27. Core Building Block - Interconnection Diagram

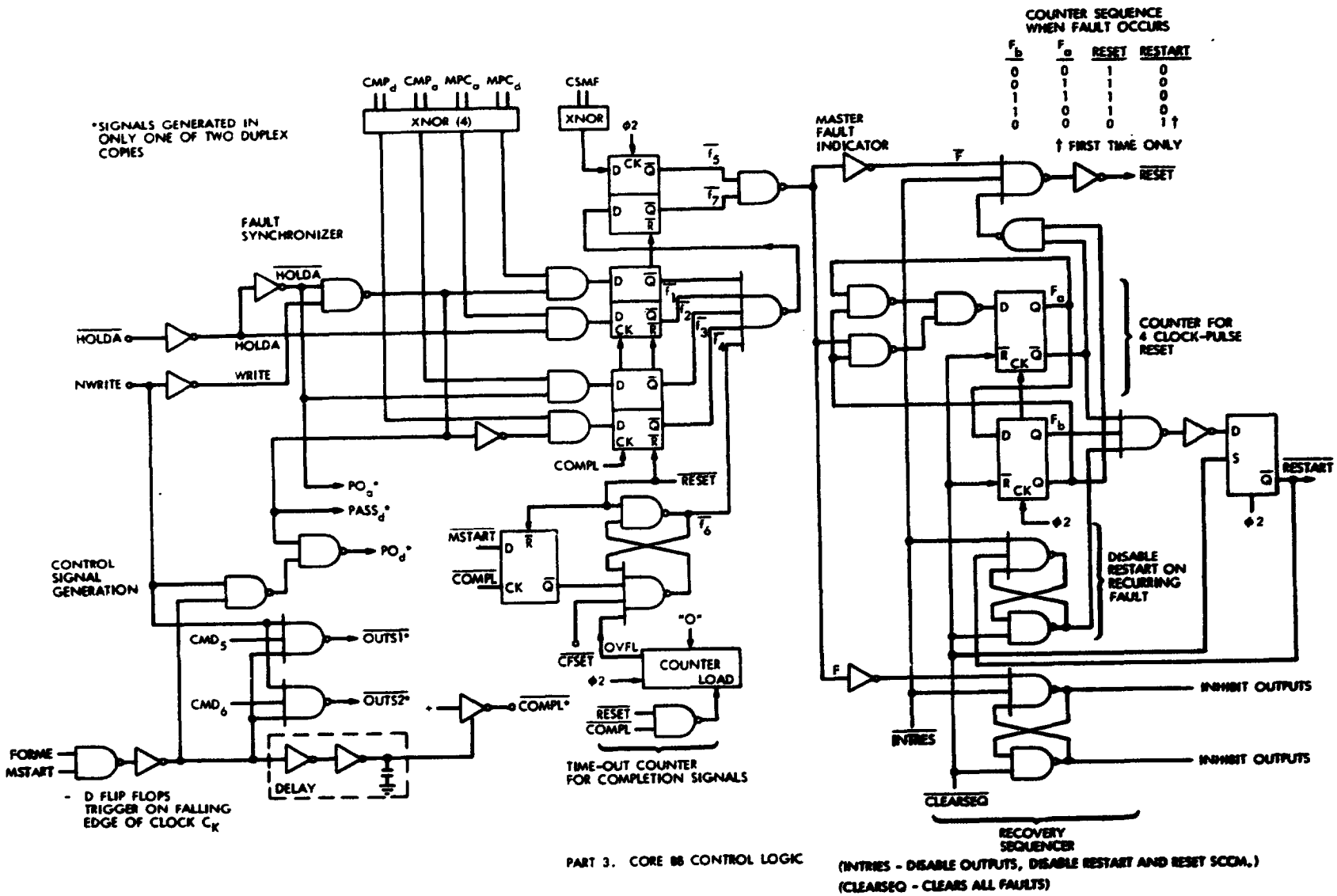


Figure 4-28. Fault Synchronizer and Recovery Sequencer

signals (CMP_d, CMP_a, MPC_d, MPC_a, CSMF) are input to a not exclusive-or function which yields a logic 1 output whenever these signals indicate a fault by taking on values 11 or 00. These signals are "anded" with a set of signals which indicate the times at which each specific check is relevant, and the result is fed to a flip-flop which is clocked at a time when the results are stable. Conditions for examining the morphic check signals are given below in Table 4-3.

Table 4-3. Conditions for Examining Morphic Check Signals

Signals	Function	Enabling Function	Strobe	F.F.
MPC_a	- Address bus parity check	HOLDA - processor off the bus	COMPL - bus completion	f_2
MPC_d	- Data bus parity check	HOLDA + READ	COMPL	f_1
CMP_a	- Compare check CPU & master CPU outputs to address bus	$\overline{\text{HOLDA}}$	COMPL	f_3
CMP_d	- Compare CPU's outputs to data bus	$\overline{\text{HOLDA}}$•WRITE	COMPL	f_4
CSMF	- All clock synchronized morphic fault indicators	at all times	ϕ_2	f_5

A flip-flop is associated with each fault indicator which is set if a fault is observed. An additional fault flip-flop (f_6) is included, which is set if (1) the bus signals MSTART and COMPL occur in improper order, (2) too much time elapses between memory complete signals (COMPL), or (3) a program rollback is

externally commanded ($\overline{\text{CFSET}}$). All six fault flip-flops are synchronized with ϕ_1 (by f_7) and combined to produce a Master Fault Indicator F which can only be set at the rise of ϕ_1.

(b) The Recovery Sequencer (RS). Upon detecting a fault (F), the RS (1) inhibits outputs, (2) generates a four clock pulse reset signal, and (3) for the first fault, commands a program rollback/restart (see F_a, F_b sequence in Figure 4-28).

When a "first" fault occurs, the RS inhibits outputs and issues a five pulse sequence. For four clock periods, a $\overline{\text{RESET}}$ signal is generated, followed by a $\overline{\text{RESTART}}$ to the CPUs to attempt a program rollback. For subsequent faults, the outputs remain inhibited and a reset is generated, but no additional $\overline{\text{RESTART}}$ is generated. (This effectively halts the CPU upon a second fault committed while trying to roll back.)

If the rollback is successful, a program command can be issued (CMD4) which clears all fault latches (CLEARSEQ), re-enables outputs, and thus provides complete absolution for the remission of faults.

(c) Control Signal Generation. One of the two Fault Handler Elements contains a small circuit for control signal generation. Internal Control Signals are generated in the following fashion.

$PASS_d$ (Pass data to check CPU) = HOLDA + READ

(don't care)

PO_a (Generate address parity) = $\overline{\text{HOLDA}}$

PO_d (Generate data parity) = $\overline{\text{HOLDA}}$·WRITE

+ READ·FORME·MSTART

$OUTS_1$ = READ·FORME·MSTART·CMD5

$OUTS_2$ = READ·FORME·MSTART·CMD6

COMPL (generate completion) = FORME·MSTART (delayed)

(d) Manual and External Module Control - This small circuit provides for clearing of fault latches in the Fault Handler and for initiating program restart. These can be carried out by front panel switches or under program control through out-of-range commands. Also included is a master reset switch and a facility for single-stepping the SCCM clock for test and debugging. The logic diagram is shown in Figure 4-29.

4.3 THE BUS INTERFACE BUILDING BLOCK (BIBB)

The Bus Interface Building Block provides the mechanism by which information is transferred between computer modules via the intercommunications bus system. The BIBB can be programmed as a Bus Adaptor or as a Bus Controller, as previously described in Section 3.5.4. The following sections provide a more detailed description of the requirements, functions, and implementation of this building block.

4.3.1 Bus System Requirements

The choice of a bus system for the fault-tolerant building-block computers requires careful consideration of functional characteristics so as to meet a wide range of applications, which is to say that, it must be useful as well as fault-tolerant. Therefore, the following general characteristics have been provided in the bus system.

(1) Formats. The Building Block Bus System (BBBS) utilizes 1553A formats to maximize compatibility with planned and existing equipments. This also defines speed and electrical characteristics. The 1553 format contains status messages required for fault-tolerant implementation.

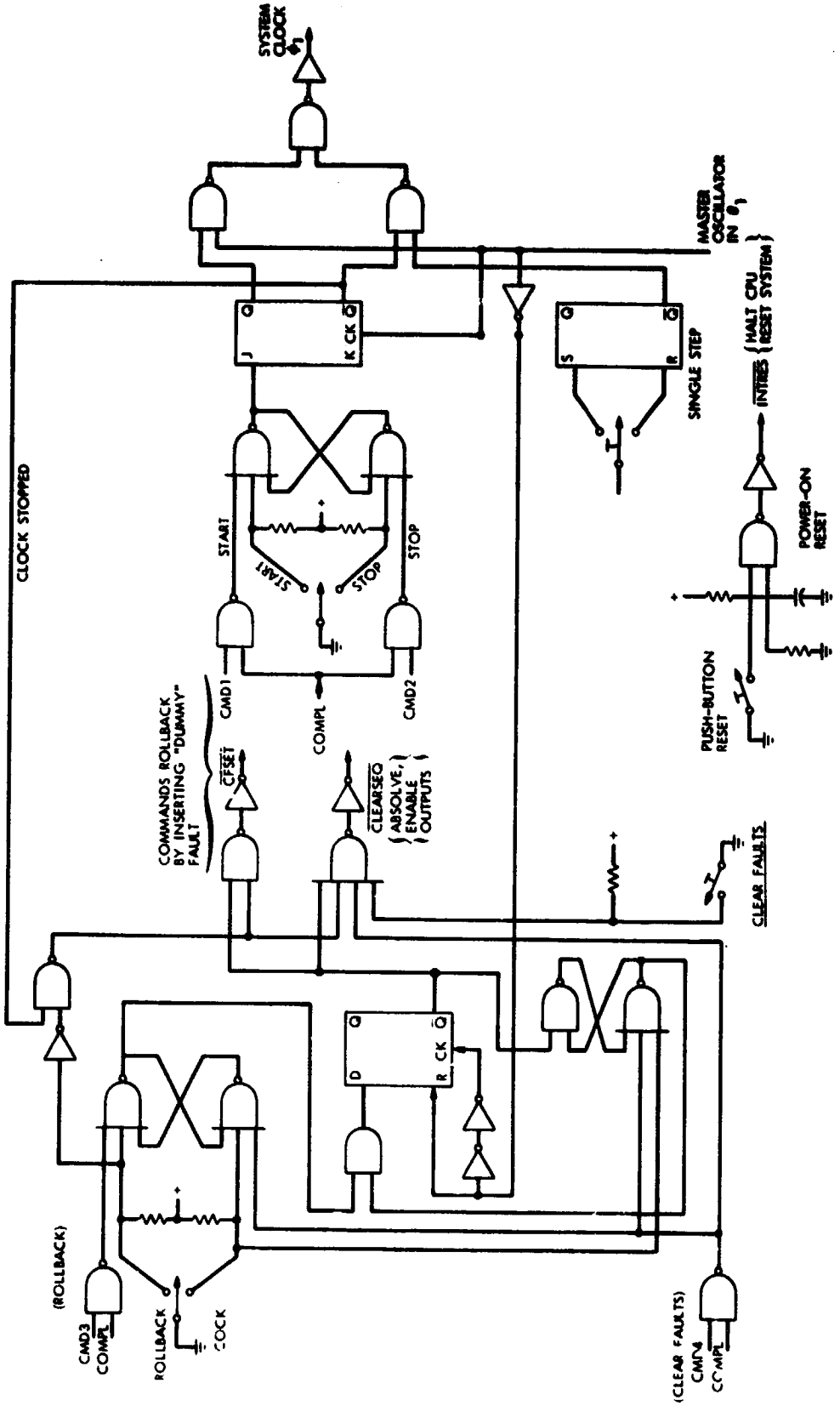


Figure 4-29. Manual and External Module Control

(2) Memory-to-Memory Transmission. The bus system is capable of moving data blocks directly between memories of the connected computers, using cycle-stealing techniques to minimize software support requirements.

(3) Indirect Addressing. Within each SCCM of a network, various areas of memory are reserved for incoming or outgoing information blocks. These data blocks can be reached through the bus system by absolute memory address or by "name" through indirect addressing.

In the first case, a typical bus command is "Move a 5-word block from source SCCM 5 location 200, to acceptor SCCM 3 location 3000." In the second case, the bus command would be, "Move 5 words from source SCCM 5, pointer 1 to acceptor SCCM 3, pointer 2."

In the indirect addressing case, the computers maintain a pointer table within their own memory which contains the addresses of the relevant data (and which is referenced by the BIBB). In our example the first pointer table entry in module 5 would contain the address 200 and the second pointer in module 3 would contain 3000.

Indirect addressing is important because it allows decoupling of the specification of global data blocks from the detailed assembly listings in the host SCCMs. Thus, software can be changed in one computer without affecting the data references in the other machines.

(4) Multiple Acceptors. The data bus is capable of transmitting information blocks from the memory of any source SCCM to the memories of one or more acceptor SCCMs. Since multiple acceptors are not directly provided in the 1553 format, additional modules must be commanded to "listen in" on a 1553-terminal-to-terminal transmission. This preserves the 1553A

format while allowing a "broadcast" mode for distributing time and engineering measurements of general interest.

(5) Block Length. The maximum length of memory blocks transmitted between computers should be at least several hundred words in order to transfer files of collected data (for a number of information collection systems). This is implemented by allowing the concatenation of 32-word transfers (the maximum number allowed in the 1553A format). Long block transfers are implemented as a sequence of 32-word transmissions in sequence followed by a final block of less than 32 words. This chopping up of long blocks into 32-word segments is carried out by the bus system in order to preserve 1553 compatibility.

(6) Universal Hardware Interface. The bus system interface with the host processors should be sufficiently general to be applied to any of a large number of different host CPUs which may be employed. The most standard interface that we could find is memory-mapped I/O. The BIBB communicates with the SCCM through the 18-bit internal address and data buses, using direct memory access (DMA). Control of the bus system by the host CPU, occurs using out-of-range addresses (memory-mapped I/O) as commands.

(7) The Bus Controller. The Bus Controller performs the Bus Control functions associated with the 1553A format, along with the augmentations described above. The Bus Controller is given a pointer to a bus control table in the host SCCM's memory by an out-of-range store instruction. The Bus Controller extracts the control table from the memory of the host module, interprets the bus table, issues bus commands to effect

the requested block transfer, carries out data block transactions involving its own memory, and monitors status signals. The host SCCM is notified of correct or erroneous transmissions through a status message left in memory and receipt of an interrupt upon completion of the transmission.

(8) The Bus Adaptor. Each Bus Adaptor moves data into and out of its associated SCCMS memory as requested by the controller of its associated bus.

(9) Requirements for Fault Tolerance. General requirements of the bus system to ensure fault tolerance are:

(a) Protection against "party line damage" of bus shorts or a bus interface talking out of turn.

(b) Detection of errors in transmission and (i) notification of the SCCM by the Bus Controller through status messages, and (ii) providing a mechanism to allow the acceptor module to determine that it has received an incomplete or erroneous message.

(c) Detection of internal faults in the Bus Controller and notification of its host SCCM. The Core-BB disables the SCCM under this condition.

(d) Detection of internal faults in a Bus Adaptor and disabling its subsequent function. (This does not disable the SCCM since other redundant Bus Adaptors may still be functioning.)

(e) The use of redundant buses and host computers so that messages can be rerouted in case of bus failures, and computations can be relocated in case of computer failures.

4.3.2 Bus Controller Functions

The Bus Controller (BC) is activated by a store instruction to one of a set of out-of-range addresses. It uses the value of the word being stored as a pointer to a Bus Control Table in the host memory. The BC reads the Bus Control Table from memory by cycle stealing and carries out the requested transfer. The BC issues those 1553A commands necessary to execute the requested data transfer over an external bus, and monitors the associated status words to verify that the transfer was properly completed. Two additional out-of-range references can be used to reset the BC or read out status. The specific memory-mapped commands to the BC are shown in Table 4-4.

Table 4-4. Memory Mapped BC Commands

Command	R/W - Address (AD0-AD15) — Comments
(1) Execute Bus Control Table	Write to: 1111 0010 ddd0 ZZZZ AD12 = Odd Parity over (AD13-AD15) (AD13-AD15) Specifies which external bus to use for transmission DB contains address of Bus Control Table
(2) Read BC Internal Status	Read from: 1111 0010 ddd1 0001 DB ← Status Register (value of internal flip-flops)
(3) Reset BC	Write to: 1111 0010 ddd1 0010 DB ignored - BC is reset

NOTES: AB0-AB3 = 0000 -- Out of Range Address
AB4-AB7 = 0010 -- Identifies BC
AB11-AB15 -- Specifies BC command
d -- don't care

4.3.2.1 The Bus Control Table (BCT). Bus Control Tables are three or four words long and have one of two formats which are decodable from the first word in the table, as shown in Table 4-5.

Table 4-5. Bus Control Table Formats

Controller/Terminal Transmission

Word 1
- = 0 Do this table and stop
- = 1 Execute next table after completing this table

Word 2 Data Address for block in BC host SCCMs memory

Word 3 A 1553A transmit or receive command

Terminal/Terminal Transmission

Word 1
- = -32768 Do this table and stop (1000 ... 00)
- = -1 Execute next table after this one (111 ... 11)

Word 2 Data Address for block in BC host SCCMs memory (to "listen in")

Word 3 A 1553A Receive Command

Word 4 A 1553A Transmit Command

Word 1 of the control table specifies a Controller/Terminal or Terminal/Controller transmission if its most significant bit is zero, and a Terminal/Terminal transmission if its MSB is one. For a sequence of short transmissions, it is useful to place their control tables in consecutive memory locations and direct the BC to execute them all automatically. This option is provided in the following fashion: If the least significant bit of word 1 equals one, the BC will automatically execute the next Bus Control Table after successfully executing the current one.

The first word of a BCT is inspected to determine which of the two formats is employed, the remaining words are interpreted in the following fashion:

(1) Transmissions Between Controller and a Terminal

The second word specifies the address within the controller's host memory where information is to be extracted or stored. If this address is positive (i.e. less than 32768), it is treated as an absolute address. If it is negative, it is complemented by the BC and used as an indirect address, i.e. the specified location is used as a pointer to the specified data. The third word is a 1553A command to be issued to the participating terminal on the bus to which information is to be sent or received.

(2) Transmission Between Terminals

For a terminal/terminal transmission the second bus table word specifies an address to store data in memory of the BCs host SCCM. The word is interpreted as in (1) above. The BC "listens in" on the transmission between terminals and stores the information in its local memory where it may or may not be used by its host processor. The third and fourth words are the 1553A receive and transmit commands necessary to set-up the specified communication.

4.3.2.2 Status on Completion or Termination. Upon completion or error termination of a communication, the BC writes a Completion Status Word (CSW) into a fixed location in memory and generates an interrupt. The CSW specifies one of five conditions:

(1) Communication OK (COM OK)

(2) Communication complete but terminals host SCCM has shutdown (MDOWN)

(3) Requested bus not available (BNA)

(4) Error in transmission -- improper coding detected or status message not returned (COMERR)

(5) Improper activity on requested bus (BACT)

In the locations immediately following the CSW, the BC stores the address of the Bus Control Table which was executed, and any (one or two) 1553A status messages that were received. Thus up to four words of status are:

N	CSW
N+1	Bus Control Table Address
N+2	1553A Status Word*
N+2	Second 1553A Status Word* (Terminal/terminal transmission)

*Only stored if received properly

4.3.2.3 Redundant Bus Utilization. The BC can be connected to several Intercommunication Buses. Its access is granted on the basis of a priority assignment established by "daisy chain" connections for each bus. The bus access control hardware is implemented in the driver/receiver logic external to the BC. The BC passes-on the bus specification in the memory mapped command (AD12-AD15) that caused its activation. The interface electronics either connects the BC with that bus or, if it is not available returns a busy indicator (BBUSY). The bus request is latched so that, if the bus is granted, the BC maintains control over the bus subsequent to the initial transmission. (Buses can be released by specifying a transmission over bus "zero", which is non existent.)

4.3.3 Bus Adaptor Functions

The Bus Adaptor responds to 1553 transmit and receive commands directed toward its host module. It accepts or delivers the number of words specified in the Word Count field of the bus command. The functions performed internal to the bus adaptor are determined by the 5-bit sub-address/mode (S/M) field of the associated command. These functions fall into two categories: transfer functions and set-up functions.

4.3.3.1 **Transfer Functions.** Twenty-eight S/M values are interpreted as Indirect Transfer instructions, and one S/M value is reserved for the Continue function, respectively, as described below:

(1) Indirect Transfer - The S/M field specifies one of 28 pointers maintained at fixed locations within the host computer memory. When a transmit or receive command is received, the bus adaptor accesses the appropriate pointer to determine the starting address for the incoming or outgoing data. By modifying pointers the host computer programs can change the physical locations accessed through each pointer.

Sequential data words are accessed for output to the bus, or input to the host memory from the bus, using DMA-cycle-stealing techniques.

Several bus adaptors may be moving data out of or into the host memory in a time-multiplexed fashion so long as none is forced to wait beyond 20 μ seconds. (The maximum word rate of the 1553A bus.)

(2) Continue - The continue function is specified by one value of the S/M field (00011) and is used for transmission of messages longer than 32 words as well as for direct addressing. The continue function in a bus command specifies that the specified transfer should continue from where the last transfer left off in the host computer memory. Thus a long message can be broken into a series of shorter transmissions from and into concatenated memory locations.

Direct memory addressing is achieved by loading an internal address register in the bus adaptor with a special setup instruction (described below). A "Continue" transfer then moves data into or out of locations beginning at the specified physical address.

4.3.3.2 Setup Instructions. Three special Setup instructions are specified by individual S/M field values. They are (1) Direct Command, (2) Direct Address, and (3) Silent Acceptor. These are all "receive" commands with a one word data transmission (WC = 1) which contains the parameters of the specified function.

(1) Direct Command (00000) - The data word sent to the adaptor (terminal) is decoded as a discrete command. If there is no error the least significant 8-bits of the received word is output (DC) and a stroke is generated by the bus adaptor. Direct commands are used to generate interrupts, to effect power switching within the host module, and other direct control functions as required.

(2) Direct Address (00010) - The data word sent to the adaptor is loaded into an internal address register and is used as a physical address from which a subsequent transfer can enter or extract data into the host memory. This setup instruction is followed by a "Continue" transfer command to move data into or out of specified locations.

(3) Silent Acceptor (00001) - The data word sent to the adaptor specifies a "soft" name. If a subsequent receive command is sent to a module with the same identification as the temporary soft name, the adaptor "listens in" on the transmission and stores the transmitted data in its own SCCM's memory. It, in effect, becomes a covert acceptor, and does not generate a status message.

The silent acceptor mode is cancelled by any subsequent Direct Command, Direct Address, or Silent Acceptor command to the module. A silent acceptor module does not return status messages, since this is done by the module which is overtly addressed.

4.3.4 BIBB Implementation

The BIBB consists of five subelements: (1) the Mill, (2) the External Bus Interface (EBI), (3) the Internal Bus Interface (IBI), (4) the Controller (CONT), and (5) the Fault Handler (FH), as shown in Figure 4-30.

The BIBB is centered around the Mill, a small processor which includes ROM, RAM, internal registers, and an ALU. Data words in transit between the external bus and the SCCM are buffered in the Mill. It is also responsible for generating addresses for DMA, word counting, te[illegible] control words, and other processing functions required of the B[illegible]

The EBI provides the interface between the Mill and the external bus. It accepts parallel command and data words from the mill and encodes them for serial transmission over the bus. It also samples incoming manchester coded data words, performs serial to parallel conversion, makes these words available to the Mill and signals the Controller of their arrival.

The IBI provides a DMA interface through which information can be transferred between the Mill and the SCCMs memory. It contains data and address registers for buffering incoming and outgoing data and DMA request and acknowledge control logic. The IBI also contains a command decoder, used to recognize and decode memory-mapped commands to the BIBB (from the host SCCM).

The Controller generates control signals for the other subelements as a function of commands received from the external or internal (SCCM) bus and conditions sampled within the BIBB. It is microprogrammed using both a ROM and a PLA.

The various circuits within the BIBB use either error detecting codes or are duplicated and compared with self-checking checkers to provide fault detection. The Fault Handler combines these fault signals into a single morphic Internal Fault Indicator (IF). Upon detection of an internal fault, the FH terminates any ongoing transmission.

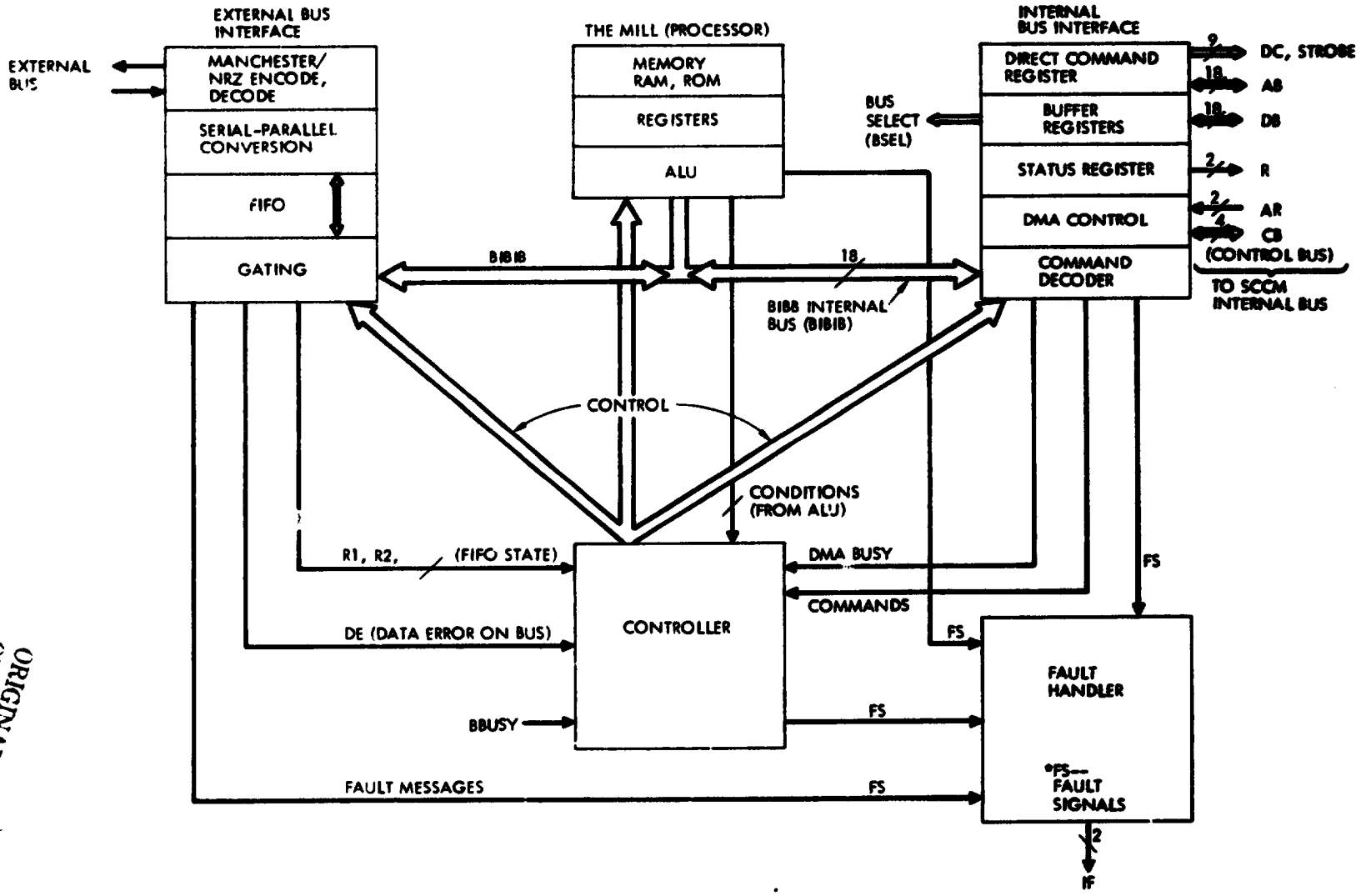


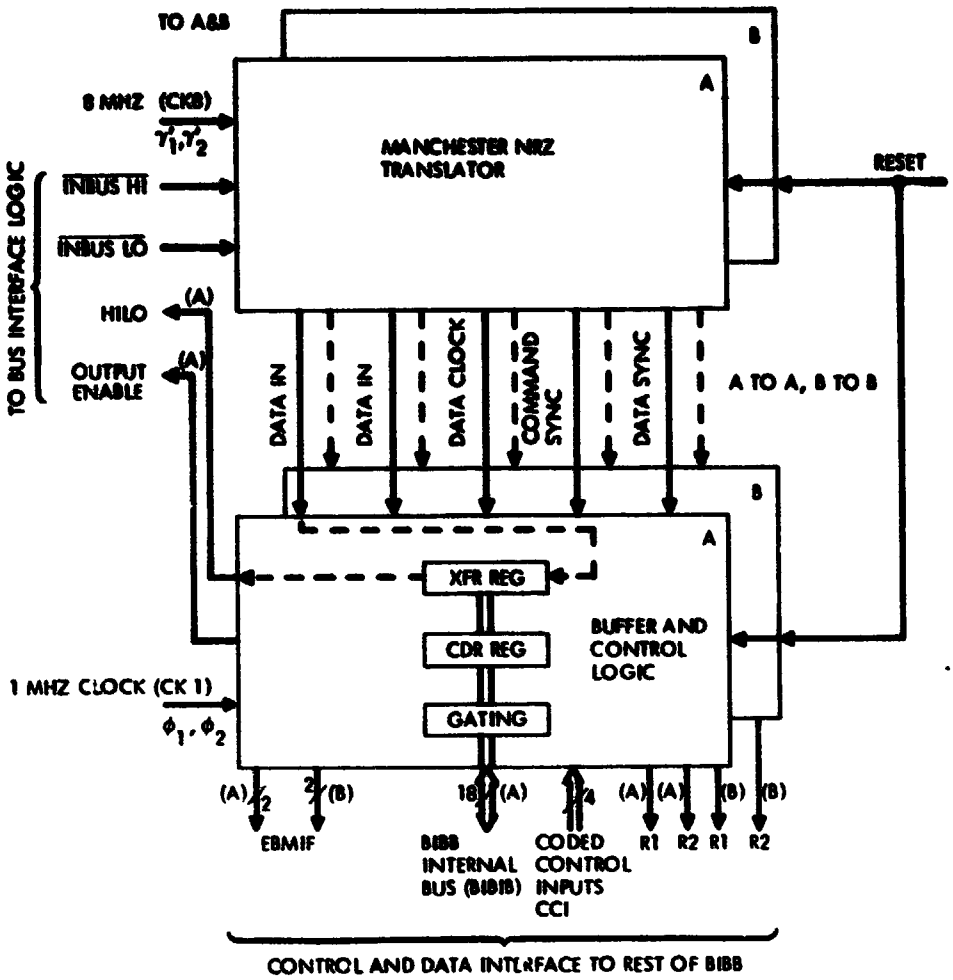
Figure 4-30. Simplified BIBB Block Diagram

In order to explain the workings of the BIBB we first examine the external and internal interface logic (EBI, and IBI). These circuits supply data and commands and largely define the environment of the Mill and the Controller. The latter two subelements are then explained as a fairly conventional processor.

4.3.4.1 The External Bus Interface. The external bus interface has two operating modes. In the input mode it decodes words appearing on the 1553A external bus, and converts these incoming serial words to parallel NRZ form. It alerts the Controller when a 1553A Command Word or Data Word has arrived, and is available for transfer to the Mill over the BIBB internal bus (BIBIB). A one-word buffer (CDR) holds an incoming command or data word while the next word may be arriving over the bus. This allows a period of 20 μsec for a word to be moved to the Mill before it is overwritten by a subsequent word arriving over the external bus. A newly arrived word in the CDR may be output to the BIBIB in three ways. The sixteen bit word may be moved directly, or if the word is a command, the word count, or S/M fields can be right justified and individually moved to the Mill.

In the output mode, words are transferred from the Mill to the EBI. Each word is designated as a command or data. A command sync or data sync is appended and the word is converted to serial biphase Manchester and output to the external bus. A one word buffer is provided in the EMI so that a new output word can be moved from the Mill to the EBI while the current word is being (serially) output. This allows up to 20 μsec to elapse between loading data words for output (before the message is interrupted for lack of data). The Controller is notified when the EBI is capable of accepting a new word, and output terminates when no words arrive from the Mill to continue the transmission.

The External Bus Interface block diagram is shown in Figure 4-31, and consists of a Manchester/NRZ Translator (MNT), and Buffer and Control Logic (BAC). The EBI is fully duplexed, i.e. there are two complete EBI circuits (A, and B) whose outputs are compared



CODED CONTROL INPUTS MNE

0	1000	NOP	
1	0001	INWRD	BIBIB ← CDR
2	0010	INWC	BIBIB (11-15) ← CDR (11-15)
3	1011	INSM	BIBIB (10-15) ← CDR (6-10)
4	0100	OUTCMD	CDR ← BIBIB, OUTPUT COMMAND TO EXTERNAL*
5	1101	OUT DATA	CDR ← BIBIB, OUTPUT DATA TO EXTERNAL BUS *

*AS SOON AS CURRENT CONTENTS OF XFR REG SENT OUT, DATA IN CDR IS TRANSFERRED TO XFR AND OUTPUT W/PRECEEDING CMD SYNC OR DATA SYNC

*OUTPUT MODE ESTABLISHED BY RECEIPT OF OUT CMD OR OUT DATA IT IS CLEARED WHEN NO FURTHER WORDS SENT FOR OUTPUT

R1	R2	MODE	
0	0	X	NO RVPT
0	1	INPUT	DATA WORD RECEIVED IN CDR FROM EXTERNAL BUS
1	0	INPUT	COMMAND RECEIVED IN CDR FROM EXTERNAL BUS
1	0	OUTPUT	CDR IS CLEAR TO ACCEPT NEXT WORD FOR OUTPUT

EBMIF - INTERNAL ERROR DETECTED IN EBM

Figure 4-31. External Bus Manager Block Diagram

to detect internal faults. A transfer register (XFR) provides serial/parallel conversion, and the Command Data Register (CDR) serves as a single word buffer through which incoming and outgoing words are passed.

Both copies of the EBM receive data from the external bus ($\overline{\text{INBUSHI}}$, $\overline{\text{INBUSLO}}$), and the BIBB internal bus (BIBIB). However, only copy A outputs data over these buses. Copy B contains morphic comparators and compares the values being output with values it is generating to defect faults.

The BIBIB is bi-directional (3-state) and consists of 18 lines. Sixteen are for data (BIBIBO-BIBIB 15) and two represent parity, using the same code as is employed in the SCCM internal bus. That is:

BIBIB 16 = $\overline{\oplus}$/ (BIBIB 0, 2, 4, ... 14)

BIBIB 17 = $\overline{\oplus}$/ (BIBIB 1, 3, 5, 7, ... 15)

Both the A and B copies of the EBI generate two control levels (R1, R2) to notify the Controller of its state. Assignments of R1, R2 are shown in Figure 4-31. In the input mode, they indicate that a command or data is available in the CDR register. In the output mode, they indicate that the CDR is free to accept new data.

Coded control inputs (CCI) to the EBI are also shown in Figure 4-31. In the input mode, they allow outputting the contents of the CDR (or only the S/M or WC) fields to the BIBIB. In the output mode they are used to load command or data words from the BIBIB into the CDR for subsequent transmission over the external bus.

The output mode is established by executing an OUTCMD or OUTDATA command. The EBI remains in the output mode until no new words are loaded into the CDR for output. It then returns to the input mode.

The following paragraphs describe a preliminary logic design of the EBM.

4.3.4.1.1 The Manchester/NRZ Translator (MNT). The MNT synchronizes with incoming bus data and delivers serial NRZ data. It also detects and signals data sync and command sync headers of the 1553A messages. The circuit, shown in Figure 4-32, has the following inputs and outputs:

INPUTS:	8 mhz clock	
	$\overline{\text{INBUSHI}}$	detect high and low
	$\overline{\text{INBUSLO}}$	levels of the 1553A bus
	RESET	sets MNT to HALT State (So)
	$\overline{\text{OUTM}}$	(NOT) OUTPUT MODE
OUTPUTS:	DATA IN	Serial bus data and 1 mhz
	$\overline{\text{DATA IN}}$	clock synchronized to
	DATA CLOCK	bus data
	DATA SYNC	Data Sync being received
	COMMAND SYNC	Command Sync being received
	$\overline{S4}$	Not in State 4

A transition and Zero Detector samples the external bus at an 8 mhz rate. If the bus has value zero during any two samples (i.e. for 125 μsec), it is assumed to be quiescent (BZRO). If the bus changes value between any two samples a transition is signalled (XTN). These signals control a simple sequencer (shown in Figure 4-32), which runs at 8 mhz.

The sequencer state is determined by 6 flip flops. The first three specify by one of six receive states (S_o - S_5) which determine the sequencer's view of what is occurring on the external bus as indicated below:

S_o - HALT	Bus is quiescent
S_1 - WAIT SYNCYC1	First Microsecond of Sync Signal (No Transitions Expected)
S_2 - RESYNC	Second Microsecond of Sync Signal (Transition Expected middle of period)

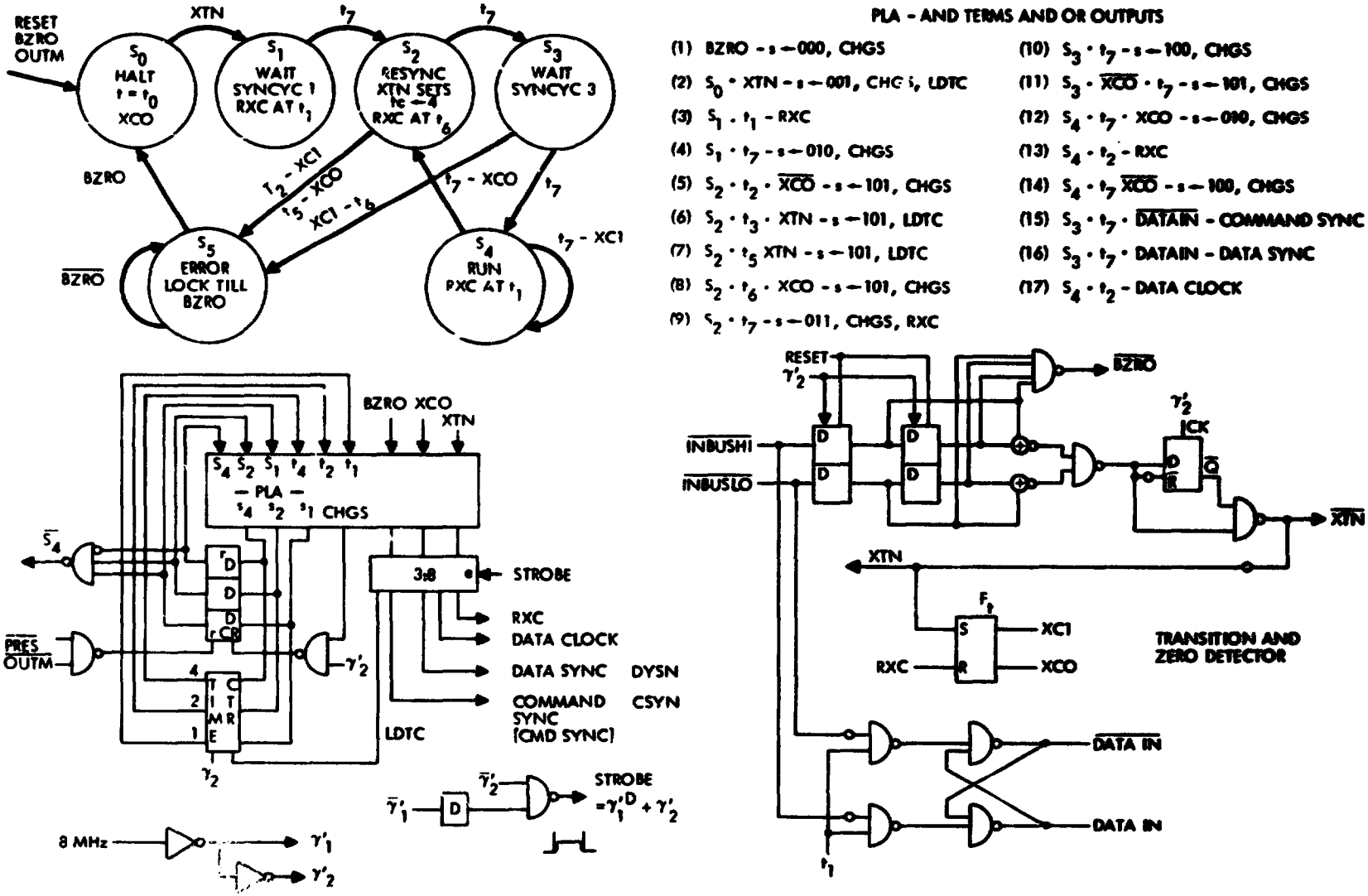


Figure 4-32. Manchester to NRZ Translator

S_3- WAIT SYNCYC3	Third Microsecond of Sync Signal (No Transitions Expected)
S_4- RUN	Data Bit Being Received (Transition Expected in middle of period)
S_5- Error	Error Detected - Stop decoding until bus becomes quiescent

The other state flip flops form a time counter which is synchronized with the incoming data. The incoming data is being received at 1 mhz, while the time counter operates at 8 mhz - counting from t_o- t_7. When the sync signal arrives, the time counter is set to t_o, and it is periodically resyncronized during bus transitions so that t_o-t_7 define one bit time on the external bus.

A state diagram for the MNT is also shown in Figure 4-32. If transitions occur on the external bus at the proper times for a "correct" transmission the states (S_o-S_4) reflect whether sync or data is being received. If an improper transition occurs, or an expected transition fails to occur in the external bus the sequencer enters the error state (S_5) and ceases operation until the current external bus transmission completes and the bus returns to zero.

The Data Clock, Data Sync, and Command Sync are derived from the sequencer and are generated during the states for which they are valid, (as shown in the state equations in Figure 4-32). A special flip flop F_t is included to "remember" if a transition has occurred on the external bus during the current bus period (1 μsecond) and is used to detect unexpected (error) transitions or lack of expected transitions on the external bus. XC1 indicates that a transition occurred, XC0 is its compliment. A special strobe pulse is generated to insure that the 3:8 decoder is only enabled when its input signals are stable. Three conditions asynchronously reset the MNT, they are external RESET, BZRO (the bus returns to zero), and OUTM. When the BIBB is outputting to the external bus (OUTM), the MNT is disabled since it is only used for input.

4.3.4.1.2 The Buffer and Control Logic (BAC). The BAC logic consists of three parts: (1) BAC Control, (2) BAC Data Paths, and (3) BAC Fault Detection Logic. The BAC is unusual in that it uses the SCCM clock (ϕ2) in the output mode (OUTM), and it uses an external-bus derived data clock for internal control in the input mode (INM).

BAC Control

Figure 4-33 shows the BAC Control logic. These circuits decode incoming commands (CCI) to the EBI, control the EBI input or output mode (INM, OUTM), and provide a counter synchronized to incoming or outgoing serial data (M1-M20). The following paragraphs describe various component parts of this logic.

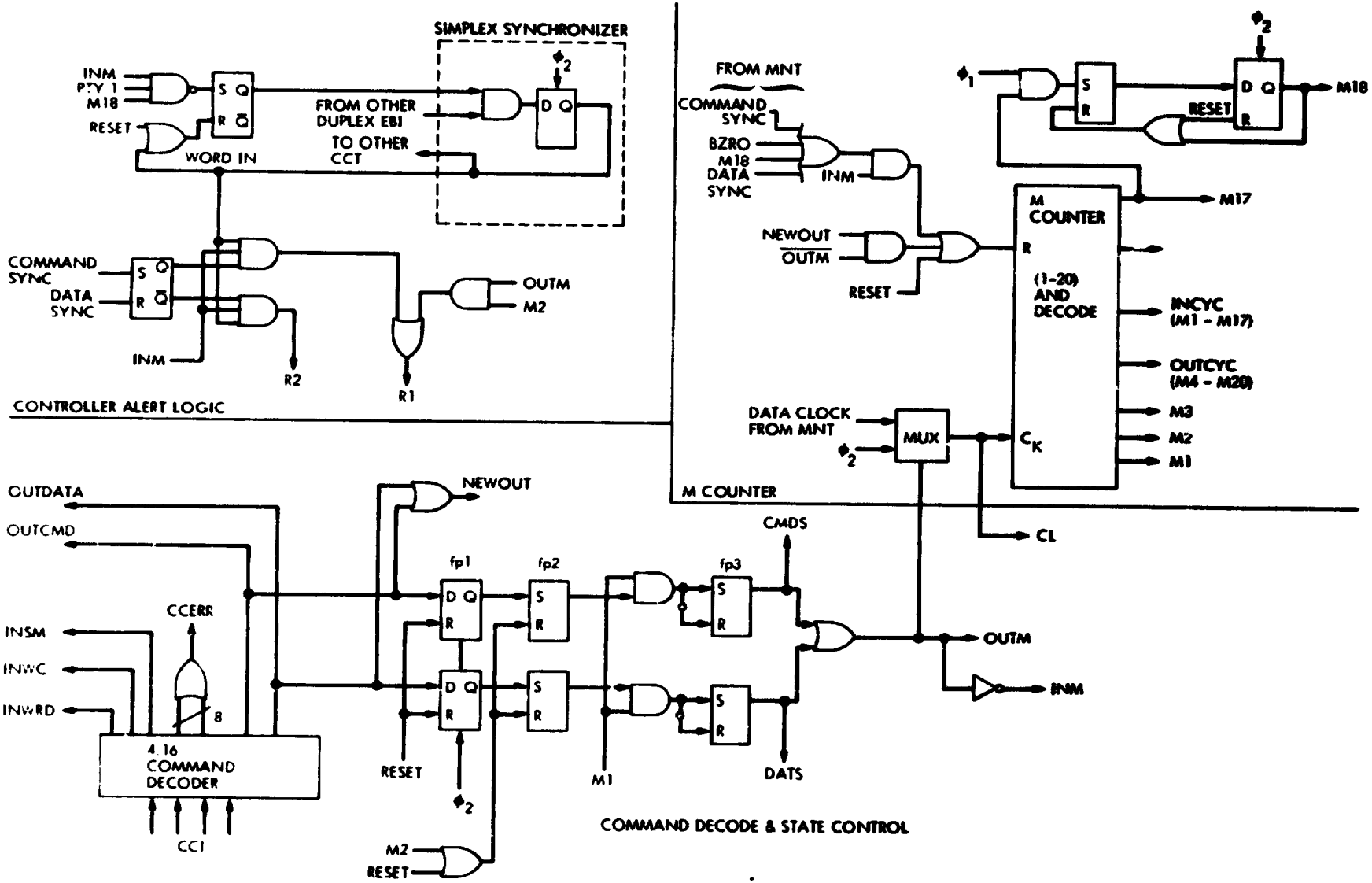


Figure 4-33. External Bus Interface, BAC - Control

(a) Command Decoder - Decodes CCI commands or detects improperly coded commands.

(b) State Control - Receipt of an OUTDATA or OUTCMD command establishes the output mode (OUTM) and causes the BAC to use the SCCM clock $\phi 2$. The OUTM mode is terminated when the EBM ready to output the next word and no new words have been sent for output, i.e., OUTCMD or OUTDATA has not been received. The three pairs of flip flops provide a means of recording the next OUTCMD or OUTDATA command (fp2) while the current such command is being executed. If no new commands have been recorded by fp2 when it is time to send out a new word (M1), fp3 is reset and OUTM is terminated.

(c) M Counter - The M Counter is synchronized with incoming or outgoing data words. In the input mode (INM) it is started at M1, when the first data bit arrives from the external bus, and reaches count M17 when the final parity bit arrives on the incoming word. During INM, the M counter is reset to M1 by an incoming Command Sync, Data Sync or no activity on the external bus. It is advanced by the incoming Data Clock which generates seventeen pulses as the data and parity bits arrive. An 18th pulse is generated (M18) to allow follow-up logic functions such as transferring the newly arrived word from the XFR register to the CDR register and alerting the BIBB Controller.

In the output mode (OUTM), the first three counts (M1-M3) designate the time when a data sync or command sync is output to the external bus. M4-M20 correspond to transmission of data bits and parity of the outgoing word. The M counter is reset by the initial OUTCMD or OUTDATA command which initiates the output mode (NEWOUT • $\overline{\text{OUTM}}$). (The M counter is a 20-count counter.)

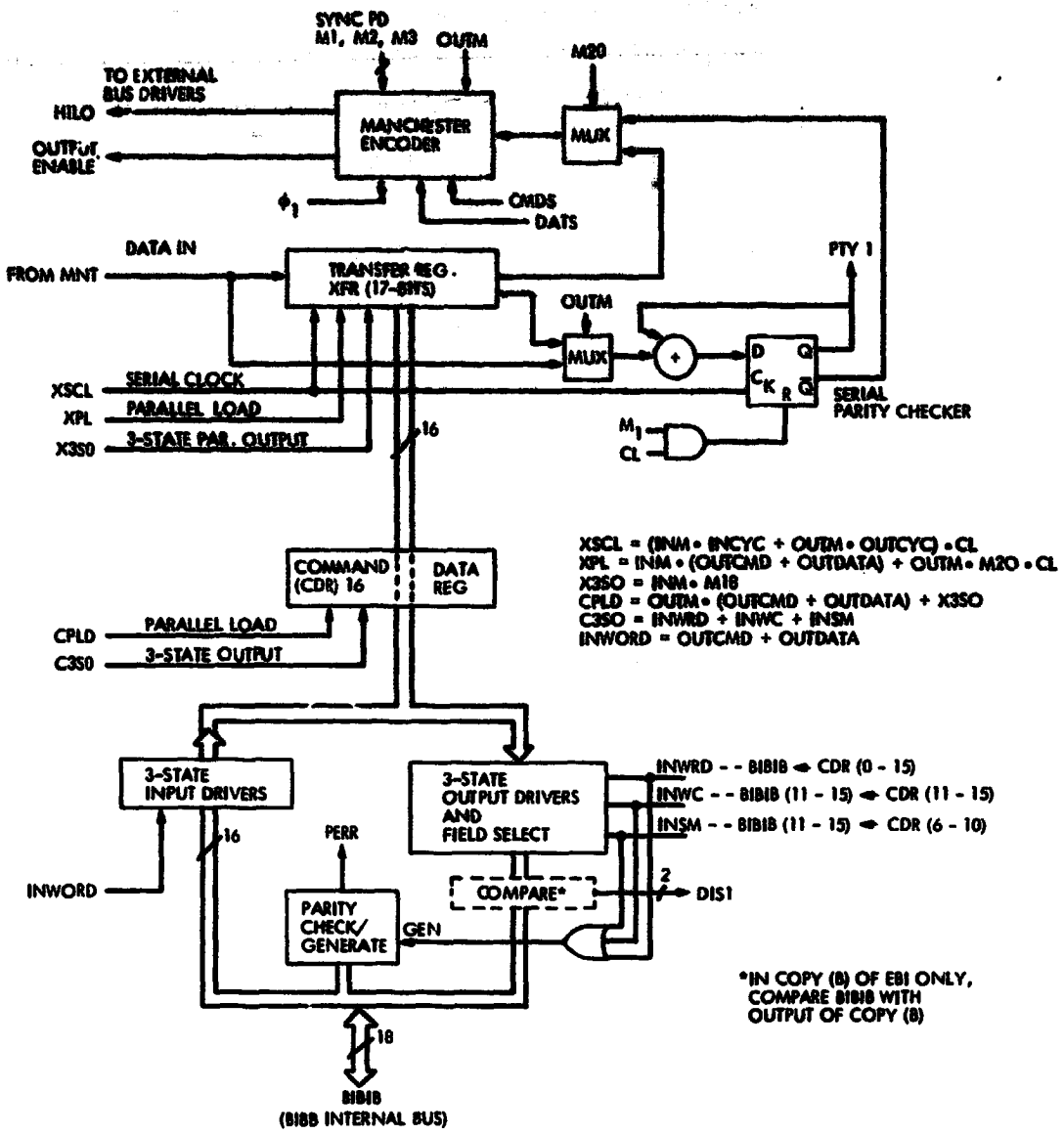
(d) Controller Alert Logic - This logic generates the signals R1,R2 which alert the BIBB Controller to the arrival of an input data word or the need for an additional output data word. This logic is straightforward with the exception of the Simplex Synchronizer.

During the input mode, an available data word is signalled by M18 and correct parity on the arrived word (IMM • PTY • M18). However, the A & B copies of the EBI may be out of step by 125 nsec since they each use their own bus-derived clock CL. The Simplex Synchronizer waits until both copies agree that the word has arrived, and then synchronizes the generation of the R1,R2 signals with the SCCM clock $\phi 2$ (which is the clock used by the BIBB Controller).

BAC Data Paths

Figure 4-34 shows the BAC Data Paths. The Transfer Register (XFR) provides serial-parallel conversion for incoming and outgoing data. A serial parity checker is used to check incoming external bus words, and to encode outgoing words. The Command-Data Register (CDR) serves as a one-word buffer between the BIBB internal bus and the XFR register. During the input mode, each incoming data word is automatically transferred to the CDR register immediately after it is assembled in XFR (at M18), and the BIBB Controller is alerted ($R2 \cdot \overline{R1}$, or $\overline{R2} \cdot R1$). The controller has approximately 19 useconds to remove the word in CDR before the next word arrives. The output driver logic allows contents of the CDR or selected fields to be output to the BIBIB.

When the output mode is established (by OUTCMD or OUTDATA) a command or data word is moved from the BIBIB to the XFR register. Subsequent OUTCMD or OUTDATA commands move data from the BIBIB to the CDR register. As each word is shipped out of the XFR (at M20) a new word is taken from the CDR register. Transmission stops when no new word is available. (It is important that the first output word not disturb the CDR. At one point in the 1553A transmission sequence, a status word is output before an 1553A command in the CDR is fully processed.)



NOTES: INM -- INPUT MODE; OUTM -- OUTPUT MODE (INM = $\overline{\text{OUTM}}$)

INCYC = M1 - M17 (COUNTS INCOMING DATA BITS AFTER SYNC)

OUTCYC = M4 - M20 (COUNTS OUTGOING DATA AFTER SYNC)

OUTCMD, OUTDATA -- CCI COMMANDS TO EBI TO TAKE WORD FROM BIBIB AND OUTPUT IT OVER EXTERNAL BUS

INWRD, INWC, INSM -- CCI COMMANDS TO OUTPUT WHOLE WORD OR FIELDS TO BIBIB

CL - CLOCK WHICH COUNTS SERIAL DATA BITS - IN OUTPUT MODE IT IS THE INTERNAL CLOCK $\phi 2$, FOR INPUT MODE IT IS A BUS-DERIVED DATA CLOCK

Figure 4-34. External Bus Interface, BAC - Data Paths

C-3

The Parity Check/Generate Circuit checks that incoming words from the BIBIB are properly coded and encodes outgoing words to the BIBIB.

The Manchester Encoder is a combinational circuit which generates a two-wire output to the 1553A bus driver, with the following interpretation:

OUTPUT ENABLE	HILO	EXTERNAL BUS
0	d	0
1	1	+1
1	0	-1

It generates a command sync (CMDS) or data sync (DATS) during M_1-M_3, and then Manchester-encodes the data bits which arrive during M_4-M_{20}.

BAC Fault Detection Logic

BAC Fault Detection Logic is shown in Figure 4-35. Each copy of the BAC compares its outputs with the other copy and, after careful strobing to assure that the signals are stable, sets a latch F1 (A,B) if they disagree. Similar latches record parity errors detected on the BIBIB (F2) and improperly coded commands (F3). In each copy of the BAC, a master fault indicator (EBMIF) is generated and sent to the BIBB Fault Handler.

Four of the fault indications (F1A, F2A, F3A, EBMIF(B)) can be sampled for diagnostic purposes by (DUMPSTAT). This function is activated by a Read Internal Status Command from the SCCM to the BIBB.

4.3.4.2 The Internal Bus Interface (IBI). The IBI provides the mechanism by which the BIBB can perform Direct Memory Access into the memory of its host SCCM. Being connected to the SCCM's internal buses, the IBI is a convenient place to place the decoding circuitry for memory-mapped commands to the BIBB.

As shown in Figure 4-36, the IBI contains three 18-bit registers to support DMA: an address register (ADROUT), and two data registers for incoming and outgoing words (DRIN, DOUT). When the BIBB sends data to the SCCM memory, it transfers an address via the BIBIB to

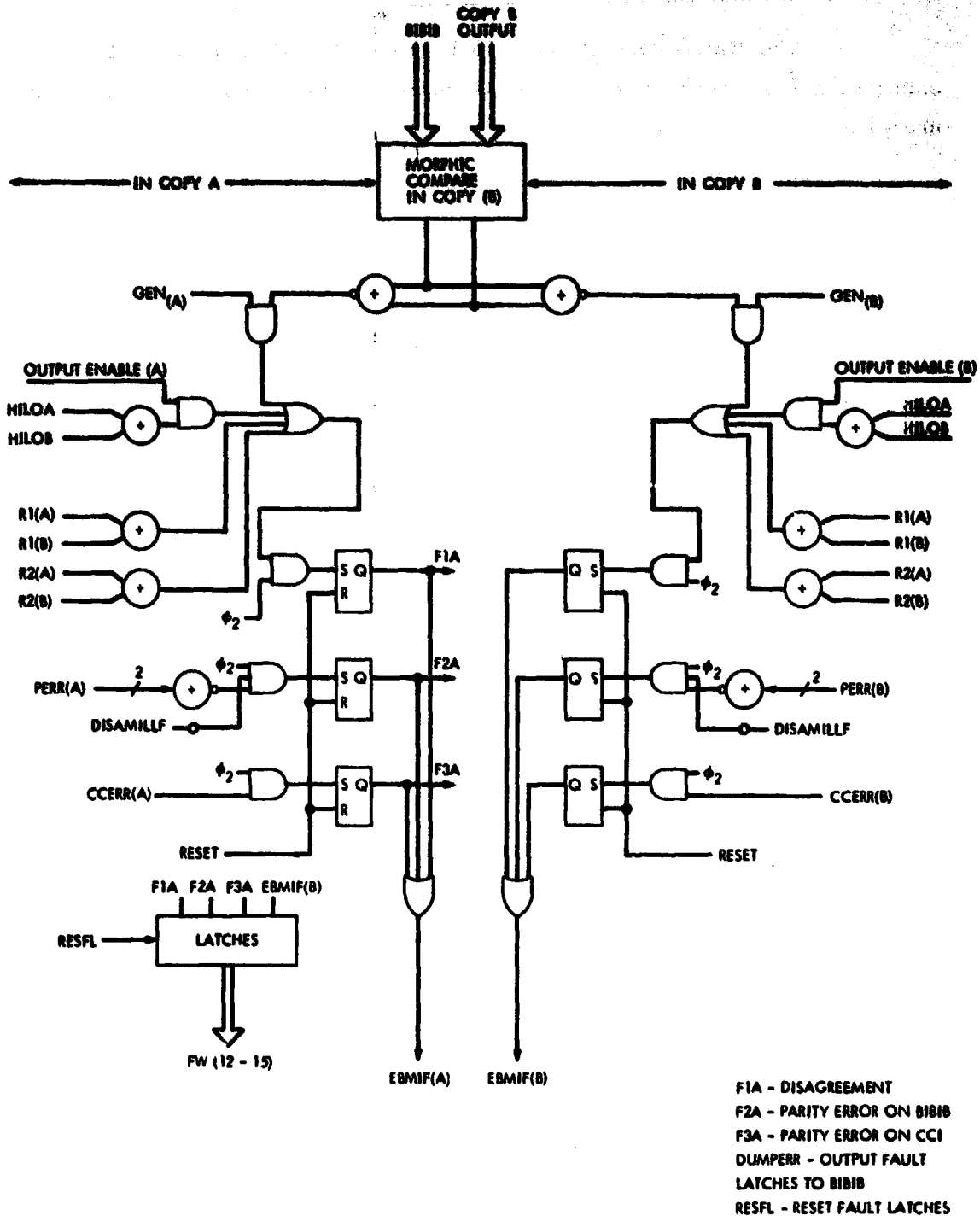


Figure 4-35. External Bus Interface, BAC - Fault Detection Logic

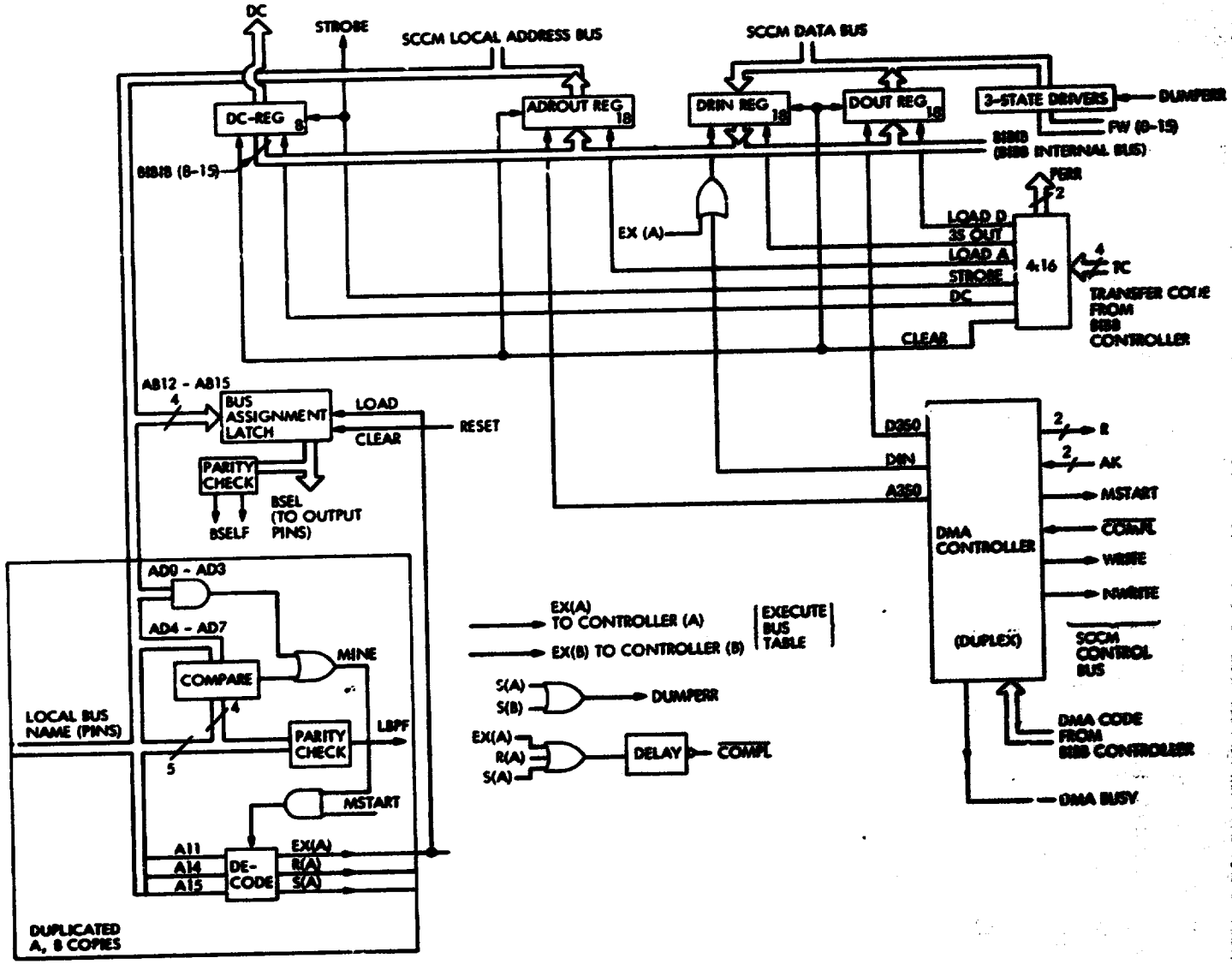


Figure 4-36. The Internal Bus Interface

ADROUT, and it transfers a data word to DOUT. The BIBB then activates the DMA Controller with a DMAWRITE command. The DMA Controller obtains control of the SCCM internal bus, and performs the specified memory write.

To read the SCCM's memory, an address is transferred from the BIBIB to ADROUT and a DMAREAD command is issued to the DMA Controller. The controller returns a busy signal while the DMA transaction is in progress.

The three address and data registers are independently controlled by the BIBB Controller and the DMA Controller. A four-bit Transfer Code (TC) is sent from the BIBB microprogrammed Controller and decoded to control transfer of data into and out of the registers from the BIBIB, as shown in Table 4-6.

An independent set of controls (D3SO, DIN, A3SO) are generated by the DMA Controller to gate data words onto or off of the SCCM local bus. Fault detection in the ADROUT, DRIN, and DOUT registers is implemented using the error detection code (with two parity bits) which is common to both the SCCM internal bus and the BIBB internal bus. In order to detect the failure mode of a disabled load signal, the registers can be periodically reset to zero (which is uncoded) by the BIBB microprogram (CLEAR).

The Direct Command Register is also included in the IBI. One form of bus transfer (DC) causes eight bits from the BIBIB to be loaded into the DC-Reg. Another command gates out this byte along with a strobe level.

Table 4-6. IBI Transfer Commands

Code	Source	Destination
0001	DRIN	BIBIB
0010	DRIN	ADROUT and BIBIB
1011	BIBIB	ADROUT
0100	BIBIB	DOUT
1101	BIBIB (8-15)	DC REG
1110	- - - - - STROBE - - - - -	

Two duplicated command decoders are employed to detect the three memory-mapped commands to the BIBB. Either decoder can issue a RESET or Read Internal Status (DUMPERR) command. Each Execute Bus Table Command is sent to one of two duplicated control sequencer circuits. If they disagree a massive disruption of control will occur and be detected in the Controller. The Bus Assignment Latch stores the number of the external bus being requested for a transmission. It is parity checked and a fault latch is set when the parity signal is stable (BSELF). Figure 4-37 shows the DMA Control Logic. Its input command codes (DMAC) are listed in Table 4-7.

The DMA Controller is an asynchronous circuit. Upon receiving a (DMAC) command, the corresponding flip flop (READ, WRITE, HOLD) is set. The SCCM internal bus is requested (R), and upon receiving an acknowledgement (AK), the following occurs:

(a) For a READ command

(1) The address is gated out (A3SO); NWRITE is raised, and a memory start ($\overline{\text{MSTART}}$) is issued.

(2) Upon receipt of a completion signal from memory (COMPL), data is gated into the DRIN register (DIN) and the READ flip flop is reset.

(b) For a WRITE command

(1) The Address is gated out (A3SO), DOUT is gated to the Data Bus (D3SO), WRITE is raised, and a $\overline{\text{MSTART}}$ is issued.

Table 4-7. DMA Command Codes (DMAC)

DMAC (0 - 2)	COMMAND
1 0 0	NO DMA -- Drop DMAHOLD
0 0 1	DMA READ -- DRIN ← M(ADROUT)
0 1 0	DMA WRITE -- M(ADROUT) ← DOUT
1 1 1	HOLD -- Hold SCCM Internal Bus

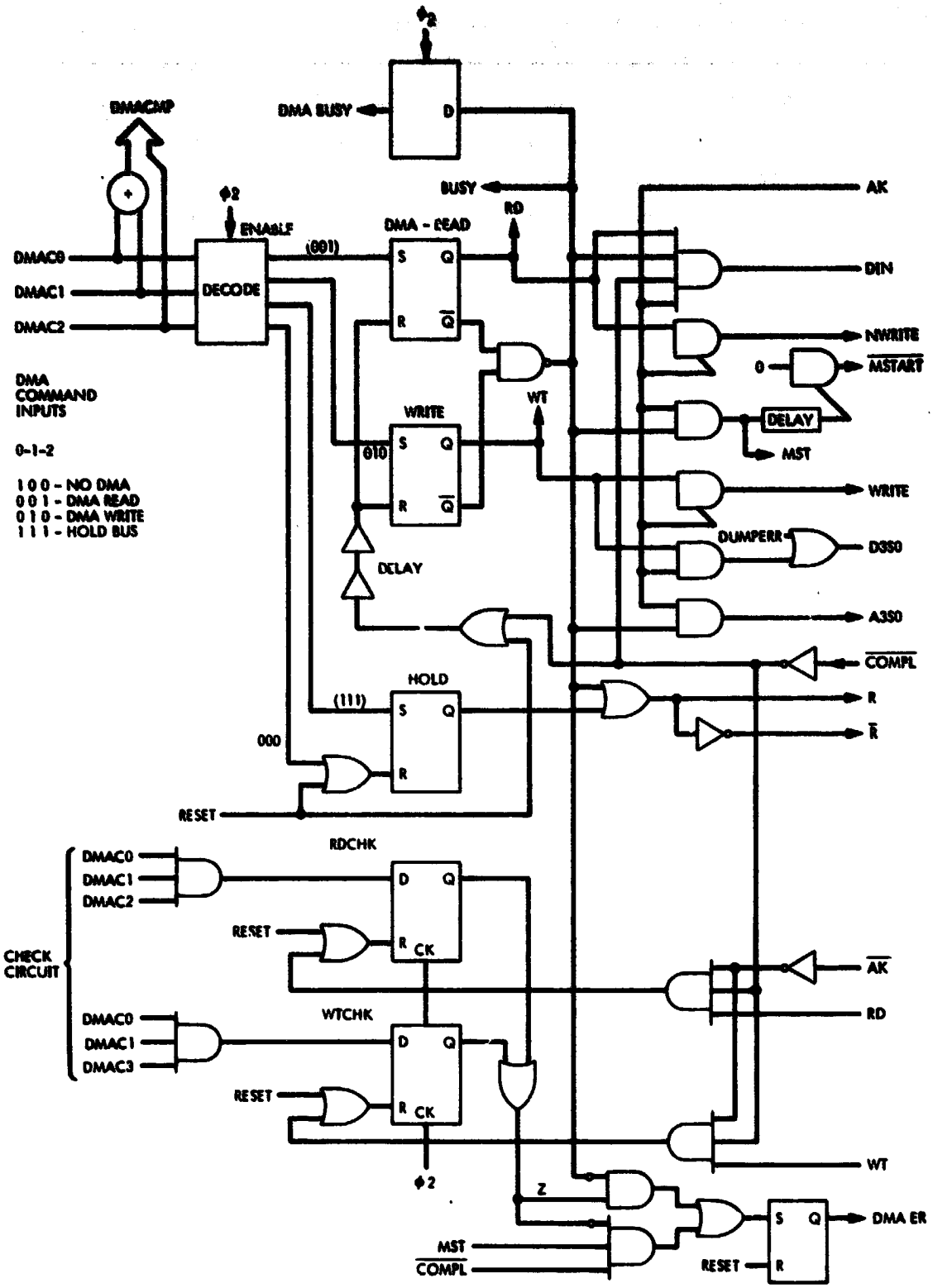


Figure 4-37. The IBI - DMA Controller

(2) Upon completion (COMPL) the WRITE flip flop is reset.

(c) The Hold state only requests (R) and holds the SCCM internal bus. Since it takes at least 3 clock periods to gain bus access, HOLD can be issued early to overlap setting up of ADROUT, DOUT, and the gaining of bus access.

The check circuit contains two flip flops which are set by READ and WRITE commands. They are reset only if they "see" that the DMA cycle actually occurred (i.e., the appropriate command level (RD, WT), a bus acknowledge (Ak) and a completion signal COMPL). Two conditions result in the fault indication DMA ER:

(1) The check circuit "saw" a DMA command but none was performed. (The check flip flops do not get reset, resulting in the $Z \cdot \overline{BUSY}$ fault condition.)

(2) A DMA was performed but the check circuits did not receive a command ($MST \cdot \overline{COMPL} \cdot \overline{Z}$).

Figure 4-38 shows the fault-handling circuitry for the IBI. There are four error checks. Both control inputs, (TC) and (DMAC) are parity encoded, and they are checked with morphic parity checkers which generate morphic signals PERR and DMACMP. These signals are synchronous with the BIBB internal clock and can be combined and sent to the Fault Handler. The other two fault signals (BSELF) and DMAER are not synchronous with the BIBB check and are latched locally within the IBI.

To generate a single "morphic" IBI fault indicator (IBIF), we reduce the two incoming morphic fault signals (PERR, DMACMP) to a single morphic pair and then logical - or the other two simplex fault signals to both lines of the morphic pair. This results in forcing the morphic pair (IBIF) to the error state (1,1), if one of the simplex fault signals is activated.

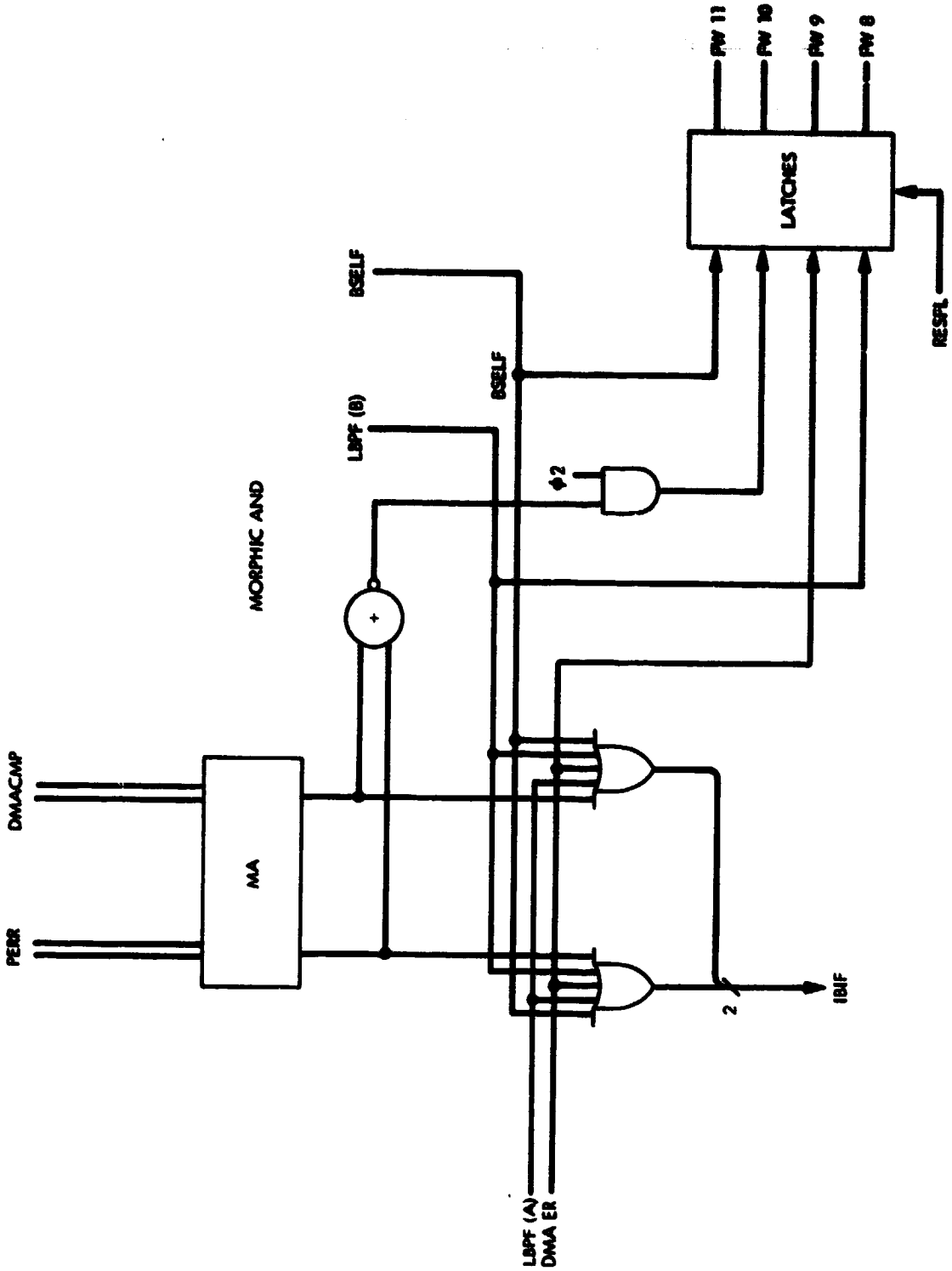


Figure 4-38. IBI - Fault Handling Circuits

Fault conditions are latched and can be read out with a read internal status command. The (DUMPERR) signal, generated by that command, causes the fault latches to be output and transferred to the SCCM data bus (see Figure 4-36).

4.3.4.3 The Mill. The Mill provides limited processing capability in the BIBB and is shown in Figure 4-39. Its two main components are a memory and ALU. The Mill memory contains 48 eighteen-bit words of RAM, and 16 eighteen-bit words of ROM. The parity encoding used to protect the BIBIB (i.e., 2 odd parity bits over even and odd bit positions) is also used to provide detection of Mill memory faults. A Mill memory word can be output to the BIBIB from an address specified by either (1) the BIBB microprogram, or (2) a local memory address register (LMADR).

Also included in the Mill are a pair of sixteen-bit A registers and ALUs. These circuits are duplicated and compared with a morphic comparator (MCALU) to implement fault detection. Words on the BIBIB can be stored in the A register and are also sent to the (b) port of the ALU. ALU outputs can be loaded back into memory or into the LMADR register. Control codes and condition codes are shown in Figure 4-39. It is possible to read modify and write a single Mill memory word in a single clock cycle (e.g., increment a location).

Four fault checks are provided which are all morphic and synchronous with the BIBB clock (ϕ_1, ϕ_2).

The address sent to Mill memory, and the BIBIB are checked for the (2 parity bit) internal bus code (MPC1, MPC2). The morphic comparison of ALU outputs produces the morphic disagreement indicator (MCALU). The control codes and memory address from the microprogram (MILC) are encoded with a single parity bit. A morphic parity check is performed producing (MILCK).

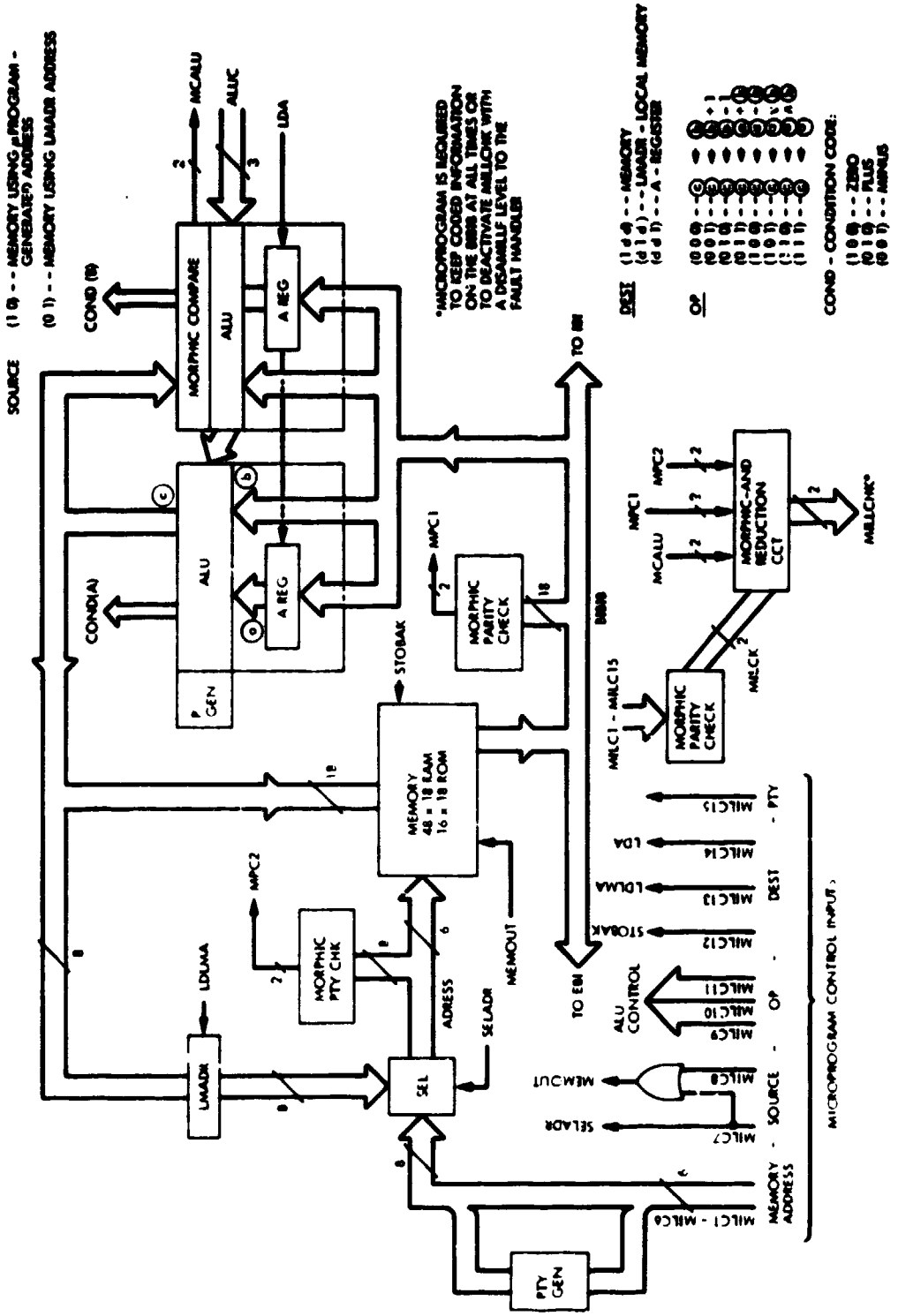


Figure 4-39(a). The Mill

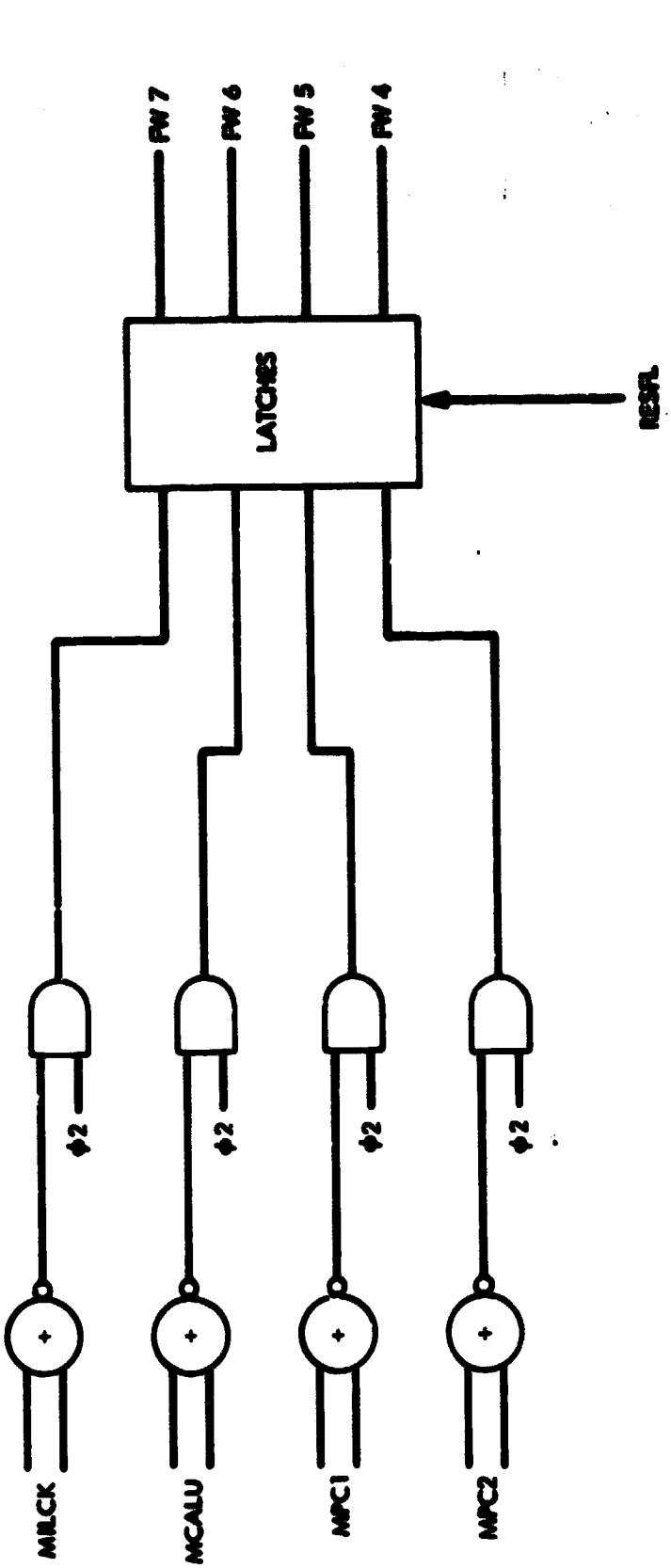


Figure 4-39(b). The Mill - Fault Latches for Status Sample

The fault indicators are combined into a single morphic fault detection pair (MILLCHK) which is sent to the BIBB Fault Handler. Figure 4-39(b) shows that the individual morphic fault indicators are decoded, latched, and made available for read out with a read internal status command to the BIBB as previously described.

4.3.4.4 The Controller. The Controller consists of a Control Sequencer (CS) and a Control ROM (CROM) which contains the microprograms for the BIBB. Figure 4-40 shows the Control Sequencer, and CROM. The CS samples various conditions from the other logic circuits within the BIBB. On the basis of these conditions it outputs a sequence of addresses to the control ROM. The CROM, in turn, maps these addresses into the control signals necessary to operate the BIBB.

Inputs to the CS are listed in Table 4-8 along with their associated control information:

Table 4-8. Control Sequencer Inputs

Input		Associated Control Information
BIBIB	-	1553A commands - Terminal I/D, S/M, and word count fields are available to CS along with T/R (transmit receive bit)
BBUSY	-	From external logic - indicates that requested bus is not available
BZRO	-	From EBI - indicates that external 1553A bus is idle
R1,R2,	-	From EBI - indicates incoming commands or data have arrived or a new word can be accepted for output (see Figure 4-31)
OUTMODE	-	From EBI - indicates EBI is in the output mode and is sending data over an external 1553A bus
COND	-	Conditions from ALU - indicate that current arithmetic result is PLUS, MINUS, or ZERO

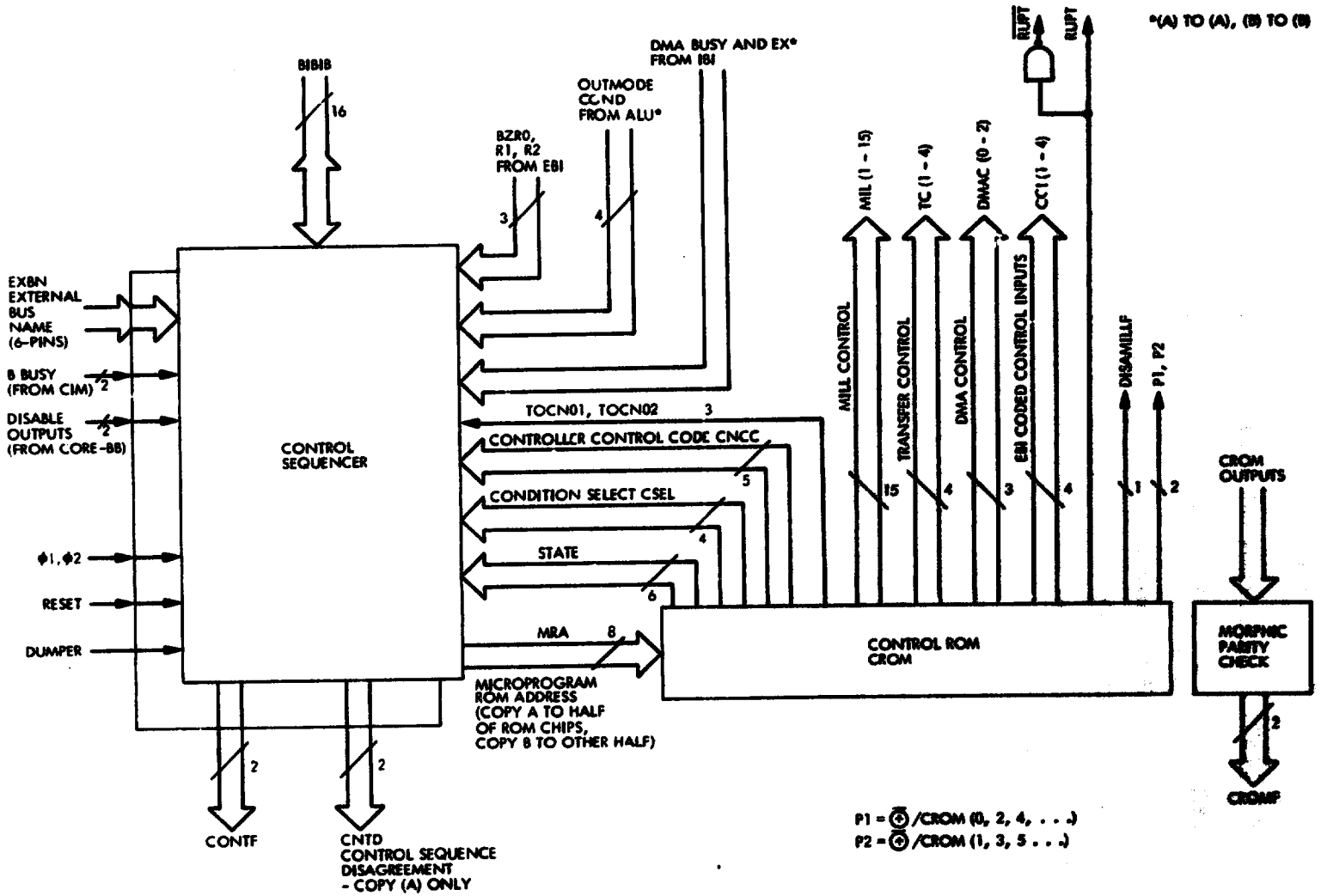


Figure 4-40. The Controller - CROM and CS

EX	-	Execute Bus Table Command received from the IBI
DMA BUSY	-	From IBI - DMA in progress
EXBN	-	From input pins - indicates "hard" name

The CROM generates a set of control levels (STATE, CSEL, CNCC, TOCNO1, TOCNO2) which are used in the CS as will be described below. Most of the other CROM outputs are the signals (previously described) which control the MILL (MILC), IBI (TC, DMAC), and EBI (CCI). Three additional signals are generated which require explanation. One (RUPT) is a programmed interrupt to the SCCM. It is pseudomorphic in that its complement is directly generated as shown. All CROM outputs are encoded in the error detecting code shown in Figure 4-40, are protected with two parity bits (P1,P2), and are checked with a morphic parity checker. One odd control is included (DISAMILLE) - disable Mill Fault Indicator. The Mill fault indicator (MILLCHK) is only valid when there is properly coded data on BIBIB, which is most of the time. For a few microinstructions, BIBIB is not coded, and the programmer commands the Fault Handler to ignore MILLCHK during these instructions.

4.3.4.4.1 The Control Sequencer (CS). The CS is built around a PLA and a Microprogram Location Counter (MLC) as shown in Figure 4-41. The MLC generates a sequence of addresses to the CROM. It is reset to zero and counts in the following fashion:

(1) If the PLA outputs a non-zero number, which is not 2^8-1 (all ones), that number will be loaded into the MLC as a branch address (executed at the next clock period).

(2) If the PLA outputs zeros, the MLC will continue to the next sequential count (address).

(3) If the PLA outputs (2^8-1) all ones, the current value of the MLC will be reloaded - holding it at its current value.

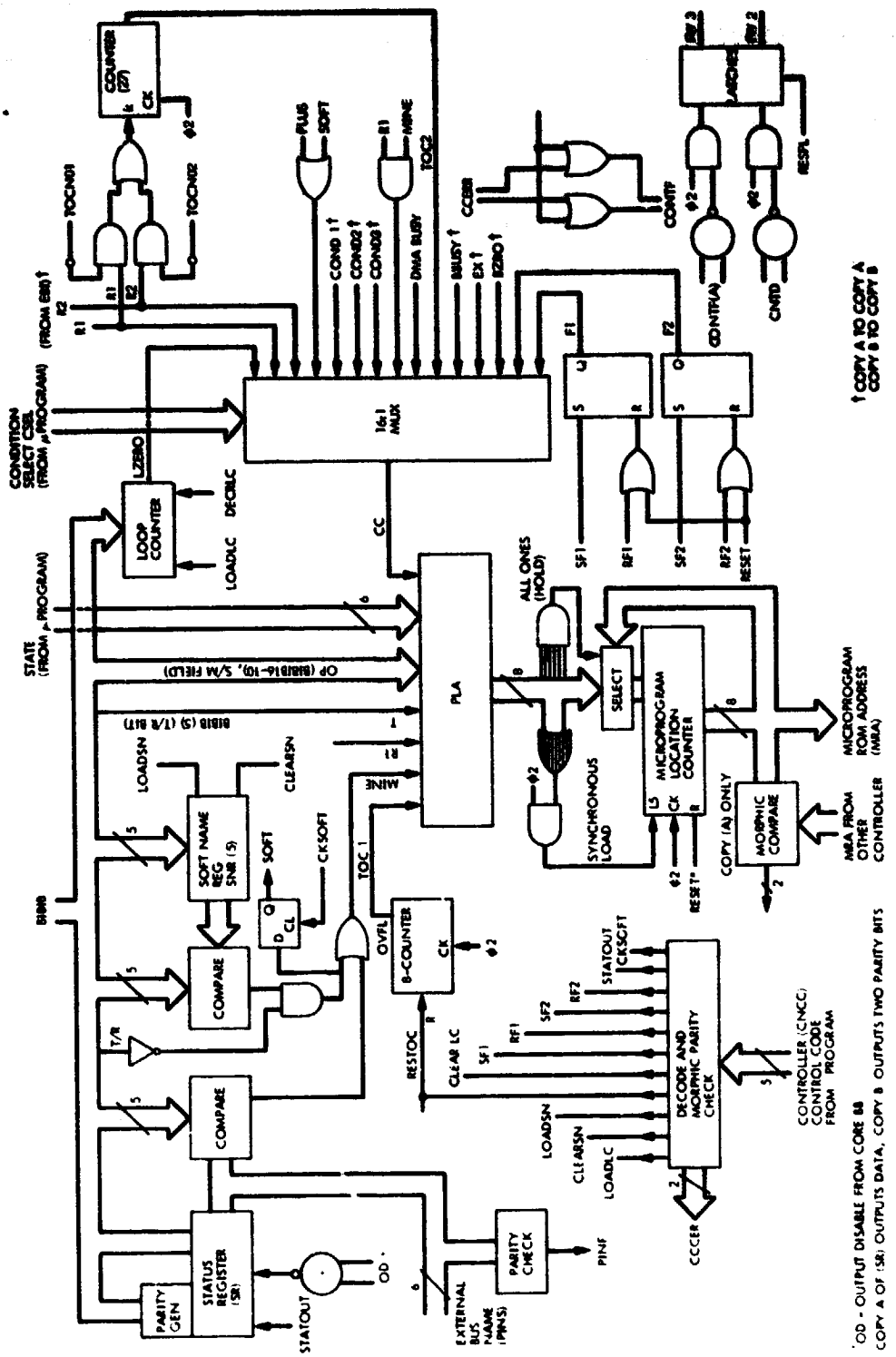


Figure 4-41. The Control Sequencer

PLA operation is controlled by the 6-bit STATE input from the microprogram. Each defined STATE input value (with the exception of state zero, S0 = 000000) activates a set of AND terms in the PLA which determine various branch addresses as a function of the PLA inputs. For State S0, no and-terms are decoded, so the microprogram proceeds sequentially.

An example of the branching technique, taken from the BUS Adaptor Microprogram is shown below in Table 4-9:

Table 4-9. A Control Sequencing Example

CROM Location	State	PLA and Terms	Control Outputs
1	S1	$\overline{R1 \cdot MINE} \rightarrow$ HOLD	BIBIB ← COMMAND
		$R1 \cdot MINE \cdot T \rightarrow 26$	
		$R1 \cdot MINE \cdot \overline{T} \cdot (OP = 0000d) \rightarrow 18$	
		$R1 \cdot MINE \cdot \overline{T} \cdot (OP = 00010) \rightarrow 18$	
		$R1 \cdot MINE \cdot \overline{T} \cdot (OP = 00011) \rightarrow 7$	
		R1•MINE and all other OP-codes fall through as sequential code	

When the microprogram gets to location one, we wish to do a five-way branch on the basis of a 1553A command received in the BIBB. We display the command on the BIBIB, which includes a T bit, and a 5-bit S/M field which is interpreted as a command OP-code. These six bits are sent directly to the PLA, along with a condition signal R1•MINE which indicates that a command has been received which was addressed to this BIBB. The state S=1 activates the five and-terms shown above.

If no command arrives (i.e., $\overline{R1 \cdot MINE}$), the PLA outputs all ones and generates a one-instruction wait loop. When the command arrives (R1•MINE), the PLA generates a transfer address corresponding to the command being decoded.

Inputs to the PLA are listed below:

(1) TOC1 - A time out counter to verify expeditious completion of DMA in the SCCM

(2) MINE - When a command is placed on the BIBIB, MINE indicates that its module ID matches the hard name or soft name of the BIBB.

(3) T - BIBIB (5) is the Transmit/Receiver bit of a 1553A command displayed on BIBIB

(4) S/M Field - BIBIB (6-10) the S/M field of a 1553A command displayed on BIBIB. It is also designated OP (for op-code to microprograms)

(5) STATE - from CROM. Activates group of PLA-AND terms to define branch(s) associated with a given micro-program location.

(6) CC - Condition signal - selected as one out of sixteen by Condition Select (CSEL) control from microprogram. Multiplexed condition signals are shown in Figure 4-41. Only one can be used at a time for a given branching instruction.

Other circuitry in the CS is explained below:

(1) Status Register (SR) - contains the 155_A status word to be output during transmissions. SR (0-4) contains the external bus name determined from the external pins. SR(5) = 1, indicates an internal fault has shut down the host SCCM, and is generated from the Output Disable levels from the Core-BB. One CS outputs data, and the other outputs the parity bits for fault detection.

(2) ID Compare - The terminal ID field of an incoming command (displayed in BIBIB) is compared with the hard name and soft name of the BIBB. If the hard name matches in a transmit command, or if either hard or soft names match in a receive command, the level MINE is raised. A soft name register can be loaded or cleared under microprogrammer control, from BIBIB (11-15). The terminal ID of zero (00000) is reserved for broadcast commands since all BIBBs with their soft

name register SNR=0 will recognise it. A latch is provided to "remember" that a soft name match occurred until the end of a transmission (SOFT). It can be reset under program control.

(3) A loop counter is provided which can be loaded from BIBIB, and decremented under microprogram control. Its underflow is signalled to the condition multiplexor LZRO.

(4) TOC1 - This time out counter counts eight pulses, and its overflow is an input to the PLA. It can be reset under program control.

(5) TOC2 - This time-out-counter is used to detect when an expected incoming or outgoing word did not arrive. It is reset by R1,R2 or both under microprogram control. TOCNO1 and TOCNO2 inhibit resetting of the counter by R1 and R2 respectively. The counter counts 26 clock times, which is longer than the time for a single word transmission. Thus if the expected words arrive, it will not overflow because it will be reset by the next arrival of a command or data sync (R1,R2) at time 20. If the expected command or data word does not arrive, the counter overflows, and delivers the signal TOC2 to the condition multiplexor.

(6) F1,F2 - These flip flops can be set, reset and tested under microprogram control.

4.3.4.5 The Fault Handler. The Fault Handler (FH) is shown in Figure 4-42. It is responsible for collecting fault signals from the BIBB and, if a fault occurs, signalling an internal fault IF. As shown in the figure, the morphic fault indicators are combined to a single morphic pair which is decoded by duplex exclusive nor circuits and combined with the EBI fault indicators (ERMIF) to set a pair of duplicated fault latches (f_1,f_2). These latches generate the IF, $\overline{IF}$ signals. These fault signals are fed to a two pairs of clocked flip flops.

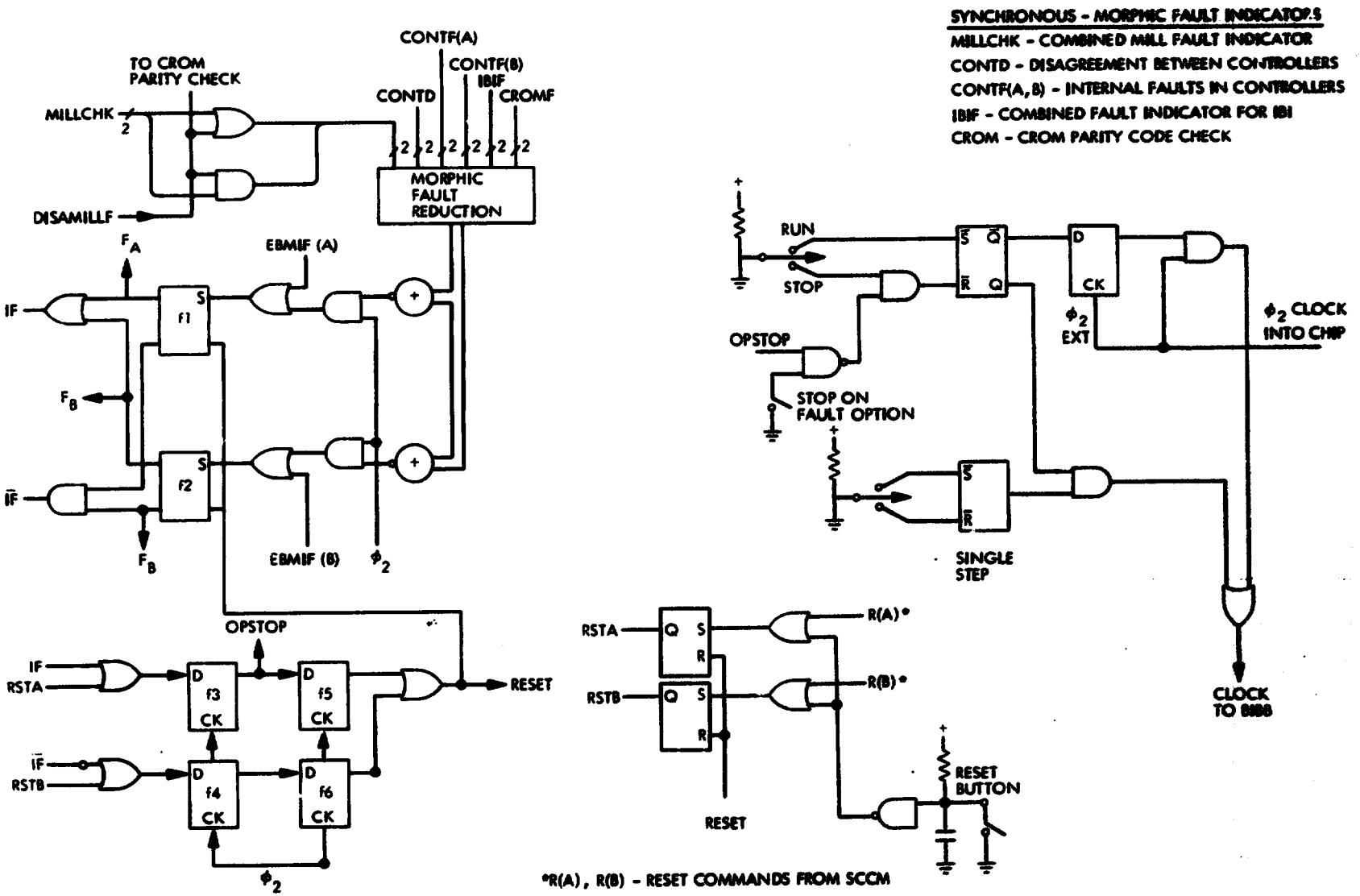


Figure 4-42. Fault Handler

The first pair (f_3, f_4) provide the option to stop the clock to the BIBB before a reset occurs. This is useful in breadboarding for fault isolation. The second pair guarantee a full cycle reset pulse to return the BIBB to an initial state.

The reset command from the SCCM generates RES(A) and RES(B) from duplex command decoders in the IBI. These simulate an internal fault which results in a reset.

Additional circuits are provided to stop, start, and single step the clock to simplify breadboarding.

4.3.5 BIBB Microprograms

The following are preliminary register-transfer descriptions of microprograms which cause the BIBB to perform as a Bus Adaptor or Bus Controller. The mnemonics refer to signals and registers previously described in this text. The notation M(XXX) refers to a Mill memory register containing the variable or constant named XXX. These variables are listed below:

(a) Bus Adaptor

BASEADR - Address in SCCM memory where the mapping table resides which maps command pointers (SM) to data addresses

PTR - A pointer used to read out or store data words in sequential locations in the SCCMS memory

WC - Word count, counts words transmitted and is taken from 1553A command field

COMND - Memory location used to store incoming command

BUSERADD - Address in SCCM where BA can store error message

ERRMESS - Error message word from the BA

(b) **Bus Controller**

BCTADR - Address of Bus Control Table in SCCM

PNT - Pointer used to access BCT words

BCT1, BCT2, BCT3
- First three words of BCT

PTR - Pointer to data words in SCCM memory

WC - Word count

STAT1 - Status word returned in a controller-terminal 1553A transmission

STAT2 - Second status word returned in a 1553A terminal-terminal transmission

STATLOC - Location where Controller Status Word is stored in SCCM memory

STATLOC+1, STATLOC+2, STATLOC+3
- Sequential locations from STATLOC

MDOWN, COMOK, COMERR, BNA, BACT
- CSW status words stored in STATLOC which indicate the results of the transmission

The Bus Adaptor and Bus Controller Microprograms are shown in Tables 4-10 and 4-11.

Table 4-10. Bus Adaptor Microprogram

CROM LOCATION		STATE	PLA-AND TERMS	CONTROL OUTPUTS
	0	S0	---	A←M(BASEADR)
		ç Load A Reg. with SCCM address of pointer table		
		ç Then wait for incoming command		
CMD	1	S1	$\overline{R1 \cdot MINE} \rightarrow HOLD$	BIBIB←CDR, CKSOFT
			$R1 \cdot MINE \cdot T \rightarrow TRMIT$	ç Transmit Command
			$R1 \cdot MINE \cdot \overline{T} \cdot (OP=0000d) \rightarrow SP$	ç Special Command
			$R1 \cdot MINE \cdot \overline{T} \cdot (OP=00010) \rightarrow SP$	ç Special Command
			$R1 \cdot MINE \cdot \overline{T} \cdot (OP=00011) \rightarrow$ WAITNEXT	ç Continue Command
		ç Branch on incoming T/R and S/M bits to processing		
		ç routine.		
RID	2	S0	---	DMA HOLD, M(PTR)←CDR(SM)+A
	3	S0	---	DMA READ, ADROUT←M(PTR)
	4	S0	---	START TOC1, M(WC)←CDR(WC)
		ç RID - Read Indirect Command -- Move WC to RAM in MILL		
		ç and start DMA cycle to get data address specified by		
		ç S/M.		
	5	S2	DMA BUSY→HOLD	
			$TOC1 \cdot \overline{DMABUSY} \rightarrow TIMEOUT$	
	6	S0	---	M(PTR)←DRIN, NO DMA
		ç We now have absolute SCCM address for incoming data		
		ç Now we wait for the data, or a XMIT command if this		
		ç is the first command of a terminal to terminal		
		ç transmission.		
WAITNEXT	7	S3	$\overline{TOC2} \cdot \overline{R1} \cdot \overline{R2} \rightarrow HOLD$	BIBIB←CDR,
			$R1 \rightarrow MAYBE$	ç maybe T/T Transmission
			$TOC2 \cdot \overline{R1} \cdot \overline{R2} \rightarrow TIMEOUT$	

Table 4-10. Bus Adaptor Microprogram (Continuation 1)

CROM LOCATION		STATE	PLA-AND TERMS	CONTROL OUTPUTS
	7		¢ If command go to Maybe, If data interrupt continue.	
	8	S4	→DATAIN+1	A first data word in CDR
DATA IN	9	S5	$\overline{R2}\cdot\overline{TOC2}$→HOLD	¢ wait for next data word in
			TOC2·$\overline{R2}$→TIMEOUT	TOCNO1
	10	S6	PLUS + SOFT→*+2	M(WC)←M(WC)-1
			¢ If soft name or not end of message skip status output.	
	11	S0	---	BIBIB←SR, OUTCMD
	12	S0	---	DMA WRITE ADROUT←M(PTR) M(PTR)←M(PTR)+1
	13	S0	---	DOUT←CDR
			¢ Write Received Word into SCCM's memory.	
	14	57	ZERO→CMD-1	M(WC)←0
			¢ If word is not zero, end message, wait for next command	
	15	S8	→DATA IN	¢ else wait for next data
MAYBE	16	S9	MINE→ERROR	BIBIB←CDR
			¢ It is a terminal/terminal transmission if not mine.	
	17	S10	$\overline{R1}\cdot\overline{TOC2}$→HOLD	TOCNO2
			R1→DATA IN	¢ When status arrives, ¢ then wait for first
			TOC2·$\overline{R1}$→TIMEOUT	¢ data word.
SP	18	S0	---	M(COMND)←CDR, CLEAR SN
	19	S5	$\overline{R2}\cdot\overline{TOC2}$→HOLD	TOCNO1
			TOC2·$\overline{R2}$→TIMEOUT	¢ If no data, timeout.
			¢ Wait above for data to arrive (R2)	

Table 4-10. Bus Adaptor Microprogram (Continuation 2)

CROM LOCATION		STATE	PLA-AND TERMS	CONTROL OUTPUTS
	20	SO	---	¢ Wait for time Status Out
	21	SO	---	BIBIB←SR, OUTCMD
	22	S11	OP=(00000)→DC OP=(00001)→SETSN	
	23	S12	→CMD-1	M(PTR)←CDR
			¢ This is the direct address command (00010) which loads ¢ a value into the pointer register. Command is com- ¢ pleted, return to zero.	
DC	24	S12	→CMD-1	DC←CDR, STROBE
			¢ Output Direct Command completed, return to zero, ¢ do not collect $200.	
SETSN	25	S12	→CMD-1	BIBIB←CDR, LOADSN
			¢ Set soft name completed	
XMIT	27	SO	---	DMA HOLD, M(PTR)←CDR(SM)+A
			¢ Establish ptr to address of data.	
	28	SO	---	ADROUT←M(PTR), DMA READ
	29	SO	---	M(WC)←CDR(WC),START TOC1
	30	S2	DMA BUSY→HOLD TOC1·$\overline{\text{DMA BUSY}}$→TIMEOUT	¢ Wait for DMA to complete
	31	SO		M(PTR)←DRIN, ADROUT←DRIN
			¢ Now we have the address of the data, next get the data.	
GETFW	32	SO		DMA READ, M(PTR)←M(PTR+1) START TOC1
	33	S2	DMA BUSY→HOLD TOC1·$\overline{\text{DMA BUSY}}$→TIMEOUT	

Table 4-10. Bus Adaptor Microprogram (Continuation 3)

CROM LOCATION		STATE	PLA-AND TERMS	CONTROL OUTPUTS
	34	S0	---	CDR←DRIN, OUTDATA, NO DMA
		¢ First data word sent out.		
	36	S7	ZERO→CMD-1	M(WC)←M(WC)-1, START TOC 1
		¢ Exit if only one word else go into output loop.		
LOOP	37	S13	$\overline{R1}\cdot\overline{TOC2}$→HOLD $\overline{R1}\cdot$TOC2→TIMEOUT	¢ Wait for EBI ready for ¢ next word.
	38	S0		DMA READ, ADROUT←M(PTR) M(PTR)←M(PTR)+1
	39	S2	DMA BUSY→HOLD TOC1·$\overline{\text{DMA BUSY}}$→TIMEOUT	
	40	S0	---	CDR←DRIN, OUTDATA
		¢ Send next word for transmission.		
	41	S7	ZERO→CMD-1	M(WC)←M(WC)-1
		¢ If word count = 0, transmission is complete		
	42	S14	→LOOP	
		¢ Else wait to deliver next word.		
TIMEOUT	43	S0	---	NOP
ERROR	44	S0		ADROUT←M (BUSER ADD)
	45	S0		DOUT←M (ERRMESS)
	46	S0		DMAWRITE
	47	S15	$\overline{\text{DMA BUSY}}$→HOLD	
	48	S13	→CMD-1	
		¢ Optional - Write error flag in SCCM memory upon ¢ detecting a bus error.		

Table 4-10. Bus Adaptor Microprogram (Continuation 4)

CROM LOCATION		STATE	PLA-AND TERMS	CONTROL OUTPUTS
TRMIT	49	S16	OP=(00011)→^+2	BIBIB←CDR
	50	S17	→XMIT	BIBIB←SR, OUTCMD
		¢ If not continue send status and go to XMIT.		
CONTX	51	S0	---	BIBIB←SR, OUTCMD, DMA HOLD
	52	S0		M(WC)←CDR(WC)
	52	S18	→GETFW	ADROUT←M(PTR)

Table 4-11. Bus Controller Microprogram

CROM LOCATION		STATE	PLA-AND TERMS	CONTROL OUTPUTS
	0	S0	---	RESET F1
START	1	S1	$\overline{EX}$→HOLD	¢ Wait for command from ¢ SCCM
	2	S2	BBUSY→ABEND1	M(BCTADR)←DRIN, ADROUT←DRIN
	3	S3	$\overline{BZRO}$→ABEND2	M(PNT)←DRIN+1
	4	S0	---	DMA READ
GETBCT	5	S4	DMA BUSY→HOLD	¢ Wait for first BCT word
	6	S0	---	M(BCT1)←DRIN
	7	S0	---	DMA READ, ADROUT←M(PNT) M(PNT)←M(PNT)+1
	8	S4	DMA BUSY→HOLD	¢ Wait in 2d BCT word.
LADR	9	S26	MINUS→INDIRECT	M(BCT2)←DRIN
	10	S0		M(PTR)←DRIN
	11	S0		DMA READ, ADROUT←M(PNT) M(PNT)←M(PNT+1)
	12	S4	DMA BUSY→HOLD	¢ Wait for 3d BCT word.
	13	S0	---	CDR←DRIN, M(BCT3)←DRIN
			¢ First 3 words of Bus Control Table moved to Mil memory ¢ M(PTR) contains address of data message in SCCM ¢ 1553A command in CDR to allow getting word count.	
	14	S0	---	M(WC)←CDR(WC)
			¢ Get the word count into M(WC)	
			¢ Next decode terminal-terminal or controller-terminal.	
	15	S5	MINUS→TT	M(BCT1)←M (BCT1)

Table 4-11. Bus Controller Microprogram (Continuation 1)

CROM LOCATION		STATE	PLA-AND TERMS	CONTROL OUTPUTS
		¢ Branch if terminal to terminal, else controller/ ¢ terminal.		
	16	S6	T→XMT	BIBIB←M(BCT3)
		¢ Branch if terminal is to transmit to controller.		
	17	S7	→REC	
		¢ Transmission from controller to terminal.		
REC	18	S0		DMA READ, ADROUT←M(PTR) M(PTR)←M(PTR)+1
	19	S4	DMA BUSY→HOLD	¢ Wait for DMA of 1st data ¢ word.
	20	S0	---	CDR←M(BCT 3), OUTCMD
		¢ OUTPUT 1553A Receive Command.		
	21	S0	---	CDR←DRIN, OUTDATA
		¢ Output first data word.		
SYNCOUT	22	S8	R1→21	¢ Wait until Data is going ¢ out.
	23	S9	ZERO→GETST	M(WC)←M(WC)-1
		¢ If this is the last word wait for status.		
	24	S0	---	DMA READ, ADROUT←M(PTR) M(PTR)←M(PTR)+1
	25	S4	DMA BUSY→HOLD	¢ Wait for DMA.
	26	S10	→SYNCOUT	CDR←DRIN, OUTDATA
GETST	27	S11	R1•TOC2→HOLD R1•TOC2→ABEND3	TOCNO2
	28	S12	→CTOUT	M(STAT1)←CDR
		¢ ABEND3 - no status received.		

Table 4-11. Bus Controller Microprogram (Continuation 2)

CROM LOCATION		STATE	PLA-AND TERMS	CONTROL OUTPUTS
XMIT	29	S0	---	CDR←M(BCT3), OUTCMD
		¢ Output 1553 Transmit Command.		
	30	S13	$\overline{R1}$→HOLD	¢ Wait for last output ¢ cycle.
		¢ Now we drop back into the input mode.		
WAITSTAT	31	S11	$\overline{R1}\cdot\overline{TOC2}$→30	TOCNO2 ¢ Wait for status.
			TOC2·$\overline{R1}$→ABEND3	¢ No status.
	32	S0	---	M(STAT1)←CDR ¢ Save status 1.
NXTDAT	33	S14	$\overline{R2}\cdot\overline{TOC2}$→HOLD	TOCNO1 ¢ Wait for data.
			TOC2·$\overline{R2}$→ABEND3	Data missing
	34	S0		ADROUT←M(PTR) M(PTR)←M(PTR)+1
	35	S0	---	DMA WRITE, DOUT←CDR
	36	S15	ZERO→OUT	M(WC)←M(WC)-1
	37	S16	→NXTDAT	
		¢ Above, Input Data Word, if WC=0, end ¢ else wait for next word.		
TT	38	S0	---	DMA READ, ADROUT←M(PNT) M(PNT)←M(PNT)+1
	39	S4	DMA BUSY→HOLD	¢ Wait for DMA of second ¢ command.
	40	S0	---	CDR←M(BCT3), OUTCMD
	41	S0	---	CDR←DRIN, OUTCMD
		¢ Output Receive CMD followed by XMT command.		
	42	S0	---	¢ Wait one period.

Table 4-11. Bus Controller Microprogram (Continuation 3)

CROM LOCATION		STATE	PLA-AND TERMS	CONTROL OUTPUTS
	43	S17	→WAITSTAT-1	SET F1
OUT	44	S18	$\overline{F1}$→CTOUT	
	45	S11	$\overline{R1}\cdot\overline{TOC2}$→HOLD TOC2·$\overline{R1}$→ABEND3	TOCNO2 ¢ Wait for second ¢ status message.
	46	S0	---	M(STAT2)←CDR ¢ Save Status 2
	47	S19	→TTOUT	
TTOUT	48	S0	---	ADROUT←M(STATLOC+3)
	49	S0	---	DOUT←M(STAT2), DMA WRITE
	50	S4	DMA BUSY→HOLD	
		¢ Write second status word for t-to-t transmission.		
CTOUT	51	S0		ADROUT←M(STATLOC+2)
	52	S0		DOUT←M(STAT1), DMA WRITE
	53	S4	DMA BUSY←HOLD	
		¢ Write first status word for t-to-t transmission.		
	54	S20	R→*+3	BIBIB←M(STAT1)
	55	S21	R→*+2	BIBIB←M(STAT2)
	56	S22	→WRCSW	DOUT←M(MDOWN)
	57	S22	→WRCSW	DOUT←M(COMOK)
		¢ If status indicates SCCM OK, skip MDOWN.		
ABEND1	58	S22	→WRCSW	DOUT←M(BNA)
ABEND2	59	S22	→WRCSW	DOUT←M(BACT)
ABEND3	60	S0	---	DOUT←COMERR

Table 4-11. Bus Controller Microprogram (Continuation 4)

CROM LOCATION		STATE	PLA-AND TERMS	CONTROL OUTPUTS
WRCSW	61	S0	---	ADROUT←M(STATLOC) DMA WRITE
	62	S4	DMA BUSY→HOLD	
	63	S0	---	DOUT←M(BCTADR)
	64	S0		ADROUT←M(STATLOC+1) DMA WRITE, OUTPUT RUPT
	65	S23	DMA BUSY→HOLD	
DONEXT	66	S0	---	A←M(ONE)
	67	S24	ZERO→START-1	M(BCT1)·A
	68	S0	---	ADROUT←M(PNT), DMA READ
	69	S25	DMA BUSY→HOLD DMA BUSY→START+1	
			¢ if first word of BCT is odd, do next table	
INDIRECT	70	S0	---	M(BCT2)←-M(BCT2)
	71	S0	---	ADROUT←M(BCT2) DMA READ
	72	S4	DMA BUSY→HOLD	---
	73	S27	→LADR+1	M(BCT2)←DRIN
			¢ if 2d word of BCT is negative,	
			¢ get indirectly specified address	

BIBLIOGRAPHY

ANDE 67 J. E. Anderson and F. J. Macri, "Multiple redundancy applications in a computer," Proc. 1967 Ann. Symposium on Reliability, Washington, D.C., January 1967, 553-562.

AVIZ 71b A. Avizienis, "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," IEEE Transactions on Computers, vol. C-20, No. 11, November 1971, 1322-1331.

AVIZ 71a A. Avizienis, et al., "The STAR (Self-Testing-And-Repairing) computer: an investigation of the theory and practice of fault-tolerant computer design," IEEE Transactions on Computers, vol. C-20, no. 11, November 1971, 1312-1321.

AVIZ 72 A. Avizienis and D. A. Rennels, "Fault-tolerance experiments with the JPL STAR computer," Digest of COMPCON '72 (Sixth Annual IEEE Computer Society Int. Conf.), San Francisco, CA, 1972, 321-324.

AVIZ 75a A. Avizienis, "Architecture of fault-tolerant computing systems," Digest 1975 Int. Symposium on Fault-Tolerant Computing, Paris, France, June 1975, 3-16.

AVIZ 75b A. Avizienis, "Fault-tolerance and Fault-intolerance: complementary approaches to reliable computing," Proc. 1975 Int. Conference on Reliable Software, Los Angeles, CA, April 1975, 458-464.

AVIZ 77a A. Avizienis, "Fault-tolerance and longevity: goals for high-speed computers of the future," Proc. Symposium on High Speed Computer and Algorithm Organization, University of Illinois, Urbana-Champaign, IL, April 1977.

AVIZ 77b A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault-tolerance during program execution," Proceedings 1977 Int. Computer Software and Applications Conference, Chicago, IL, November 1977.

AVIZ 77c A. Avizienis, "Fault-tolerant computing: progress, problems and prospects," Proc. IFIP Congress 1977, Toronto, Canada, 405-420.

AVIZ 80 A. A. Avizienis, "Fault-tolerance: The survival attribute of digital systems," Proc. IEEE, vol. 66, No. 10, October 1978, pp 1109-1125.

BARL 65 R. W. Barlow and F. Proschan, Mathematical theory of reliability, Wiley and Sons, New York, 1965.

BEUM 73 L. O. Beum, "Standardization of avionics information systems," System Development Corporation, Santa Monica, CA, TM-5159/000/00A, performed for ARPA Institute for Defense Analysis, August, 1973.

BEUS 69 H. J. Beuscher, et al., "Administration and maintenance plan of no. 2 ESS," The Bell System Technical Journal, vol. 48, October 1969, 2765-2815.

BORG 72 B. R. Borgerson, "A fail-softly system for time-sharing use," Digest 1972 Int. Symposium on Fault-Tolerant Computing, June 1972, 89-93.

BOUR 69 W. G. Bouricius, W. C. Carter, and P. R. Schneider, "Reliability modeling techniques for self-repairing computer systems," Proc. 24th National Conference of ACM, 1969, 295-383.

BOUR 71 W. G. Bouricius, et al., "Reliability modeling for fault-tolerant computers," IEEE Transactions on Computers, vol. C-20, November 1971, 1306-1311.

BREU 76 M. A. Breuer and A. D. Friedman, Diagnosis and reliable design of digital systems, Computer Science Press, Inc., Woodland Hills, CA, 1976.

BRIC 73 J. L. Bricker, "A unified method for analyzing mission reliability for fault-tolerant computer systems," IEEE Transactions on Reliability, vol. R-22, no. 2, June 1973, 72-77.

BUTL 74 T. T. Butler et al., "LAMP: application to switching-system development," The Bell System Technical Journal, vol. 53, no. 8, October 1974, 1535-1555.

CART 64 W. C. Carter, et al., "Design of serviceability features for the IBM system 360," IBM Journal of Research and Development, vol. 8, no. 2, April 1964, 115-125.

CART 72 W. C. Carter et al., "Computer error control by testable morphic boolean functions - a way of removing hardcore," Dig. of the 1972 International Symposium on Fault-Tolerant Computing, Newton, MA, IEEE Computer Society, June 1972, 154-159.

CART 74 W. C. Carter, "Theory and use of checking circuits," Computer Systems Reliability, Infotech Information Ltd., 1974 (Maidenhead, England), 413-454.

CART 76 W. C. Carter and C. E. McCarthy, "Implementation of an experimental fault-tolerant memory system," IEEE Transactions on Computers, vol. C-25, no. 6, June 1976, 557-568.

CART 77 W. C. Carter, et al., "Cost effectiveness of self-checking computer design," in Dig. 1977 Int. Symp. Fault-Tolerant Computing (Los Angeles, CA), pp 117-123, June 1977.

CHAN 74 H. Y. Chang, G. W. Smith, Jr., and R. B. Walford, "LAMP: system description," The Bell System Technical Journal, vol. 53, no. 8, October 1974, 1431-1449.

CONN 72 R. B. Conn, N. A. Alexandridis and A. Avizienis, "Design of a fault-tolerant modular computer with dynamic redundancy," AFIPS Conference Proc. vol. 41, Fall JCC 1972, 1057-1067.

COOP 76 A. E. Cooper and W. T. Chow, "Development of on-board space computer systems," IBM Journal of Research and Development, vol. 20, no. 1, January 1976, 5-19.

CORB 72 F. J. Corbato, J. H. Saltzer, and C. T. Clingen, "Multics: the first seven years," AFIPS Conference Proceedings, vol. 40, 1974, 571-583.

DOWN 64 R. W. Downing, J. S. Nowak, and L. S. Tuomenoksa, "No. 1 ESS maintenance plan," The Bell System Technical Journal, vol. 43, no. 5, part 1, September 1964, 1961-2019.

EJCC 53 Information processing systems - reliability and requirements, Proc. of the Eastern Joint Computer Conference, Washington, D.C., December 1953.

EVER 57 R. R. Everett, C. A. Zraket, and H. D. Benington, "SAGE - A data-processing system for air defense," Proc. Eastern Joint Computer Conference, Washington, D.C., December 1957, 148-155.

GOLD 75 J. Goldberg, "New problems in fault-tolerant computing," Digest 1975 Int. Symposium on Fault-Tolerant Computing, Paris, France, June 1975, 29-34.

HAME 72 K. J. Hamer-Hodges, "Fault resistance and recovery within System 250," Proceedings of I.C.C. Conference, Washington, D.C., October 1972.

HECH 76 H. Hecht, "Fault-tolerant software for real-time applications," ACM Computing Surveys, vol. 8, no. 4, December 1976, 391-407.

HOPK 75 A. L. Hopkins, Jr. and T. B. Smith III, "The architectural elements of a symmetric fault-tolerant multiprocessor," IEEE Transactions on Computers, vol. C-24, no. 5, May 1975, 498-505.

HOPK 78 A. L. Hopkins, Jr., et al., "FTMP - A highly reliable fault tolerant multiprocessor for aircraft," Proc. IEEE, vol. 66., no. 10, October 1978, pp. 1221-1239.

IBM 67 An application-oriented multiprocessing system, IBM Systems Journal, vol. 6, no. 2, 1967.

ICRS 75 Proceedings of the 1975 Int. Conference on Reliable Software, Los Angeles, CA, April 1975.

IRE 53 Session 14: Symposium - diagnostic programs and marginal checking for large scale digital computers, Convention Record of the IRE 1953 National Convention, part 7, New York, N.Y., March 1953, 48-71.

KATS 78 D. Natsuki, et al., "Pluribus - An operational fault-tolerant multiprocessor," Proc. IEEE, vol. 66, no. 10, October 1978, pp. 1146-1159.

LESH 76 H. F. Lesh, et al., "Software techniques for a distributed real-time processing system," in Proc. IEEE National Aerospace and Electronics Conf. (Dayton, OH), pp. 290-295, May 1976.

LEVY 75 H. O. Levy and R. B. Conn, "A simulation program for reliability prediction of fault tolerant systems," Digest 1975 Int. Symposium on Fault-Tolerant Computing, Paris, France, June 1975, 104-109.

LOND 75 R. L. London, "A view of program verification," Proc. 1975 Int. Conference on Reliable Software, Los Angeles, CA, April 1975, 534-545.

MAIS 71 F. P. Maison, "The MECRA: a self-repairable computer for highly reliable process," IEEE Transactions on Computers, vol. C-20, no. 11, November 1971, 1382-1393.

MATH 70 F. P. Mathur, and A. Avizienis, "Reliability analysis and architecture of a hybrid-redundant digital system; generalized triple modular redundancy with self-repair," AFIPS Conference Proceedings, vol. 36, 1970, 375-383.

MATH 75a F. P. Mathur, and P. T. de Sousa, "Reliability modeling and analysis of general modular redundant systems," IEEE Transactions on Reliability, vol. R-24, No. 5, December 1975, 296-299.

MATH 75b F. P. Mathur, and P. T. de Sousa, "Reliability models of NMR systems," IEEE Transactions on reliability, vol. R-24, no. 2, June 1975, 108-112.

MCPH 76 J. A. McPherson and C. R. Kime, "A two-level approach to modeling system diagnosability," Proc. 1976 Int. Symposium on Fault-Tolerant Computing, Pittsburgh, PA, June 1976, 33-38.

MERA 76 C. Meraud and F. Browaeys, "Automatic rollback techniques of the COPRA computer," Proc. 1976 Int. Symposium on Fault-Tolerant Computing, Pittsburgh, PA, June 1976, 23-29.

MOOR 56 E. F. Moore and C. E. Shannon, "Reliable circuits using less reliable relays," Journal of the Franklin Institute, vol. 262, nos. 9 and 10, September, October 1956, 191-208 and 281-297.

MORA 75 P. B. Moranda, "Prediction of software reliability during debugging," Proc. 1975 Annual Reliability and Maintainability Symposium, January 1975, 327-332.

NELS 75 E. C. Nelson, "Software reliability," Digest 1975 Int. Symposium on Fault-Tolerant Computing, Paris, France, June 1975, 24-28.

NGYW 76 Y. -W. Ng and A. Avizienis, "A model for transient and permanent fault recovery in closed fault-tolerant systems," Proc. 1976 Int. Symposium on Fault-Tolerant Computing, Pittsburgh, PA, June 1976, 182-188.

NGYW 77a Y. -W. Ng and A. Avizienis, "A reliability model for gracefully degrading and repairable fault-tolerant systems," Proc. 1977 Int. Symposium on Fault-Tolerant Computing, Los Angeles, CA, June 1977, 22-28.

NGYW 77b Y. -W. Ng and A. Avizienis, "ARIES - an automated reliability estimation system," Proc. 1977 Annual Reliability and Maintainability Symposium, Philadelphia, PA, January 1977, 108-113.

OBLO 62 J. Oblonsky, "A self-correcting computer," Digital Information Processors, W. Hoffman, ed., Interscience Publishers, New York, 1962, 533-542.

ORNS 75 S. M. Ornstein, et al., "Pluribus - a reliable multiprocessor, AFIPS Conference Proceedings," vol. 44, 1975, 551-559.

PARH 74 B. Parhami and A. Avizienis, "A study of fault-tolerance techniques for associative processors," AFIPS Conference Proceedings, vol. 43, 1974, 643-652.

PARK 76 K. P. Parker, "Compact testing: testing with compressed data," Proc. 1976 Int. Symposium on Fault-Tolerant Computing, Pittsburgh, PA, June 1976, 93-98.

RAND 75 B. Randell, "System structure for software fault tolerance," IEEE Transactions on Software Engineering, vol. SE-1, no. 2, June 1975, 220-232.

RAMA 72 C. V. Ramamoorthy and L. C. Chang, "System modeling and testing procedures for microdiagnostics," IEEE Transactions on Computers, vol. C-21, no. 11, November 1972, 1169-1183.

RENN 73a D. A. Rennels and A. Avizienis, "RMS: A reliability modeling system for self-repairing computers," Digest of the Third International Symposium on Fault-Tolerant Computing, Palo Alto, CA, June 1973, 131-135.

RENN 73b D. A. Rennels, "Fault detection and recovery in redundant computer using standby spares," Technical Report UCLA-ENG-7355, University of California, Los Angeles, CA, June 1973.

RENN 76 D. A. Rennels, et al., "The unified data system: A distributed processing network for control and data handling on a spacecraft," in Proc. IEEE National Aerospace and Electronics Conf. (Dayton, OH), pp. 283-289, May 1976.

RENN 78a D. Rennels, Fault-Tolerant building-block computer study, JPL Publication 78-67, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, July 1978.

RENN 78b D. Rennels, "Architectures for fault tolerant spacecraft computers," Proc. IEEE, Vol. 66, No. 10, October 1978, pp. 1255-1268.

RENN 80a D. A. Rennels, "Distributed Fault-Tolerant Computer Systems," Computer, Vol. 13, No. 3, March 1980, pp 55-65.

RENN 80b D. Rennels and M. Buchwalter, "Selective redundancy in a building-block distributed computing system," Dig. Government Microcircuit Applications Conference, (Houston, TX), to be published November 1980.

SHOO 73 M. L. Shooman, "Operational testing and software reliability estimation during program development," Proc. 1973 IEEE Symposium on Computer Software Reliability, New York City, 1973, 51-56.

SHOR 68 R. A. Short, "The attainment of reliable digital systems through the use of redundancy - a survey," IEEE Computer Group News, vol. 2, no. 2, March 1968, 2-17.

SIEW 77 D. Siewiorek, M. Canepa and S. Clark, "C.vmp: The architecture of a fault-tolerant multiprocessor," Proc. 1977 Int. Symposium on Fault-Tolerant Computing, Los Angeles, CA, June 1977.

SKLA 76 J. R. Sklaroff, "Redundancy management technique for space shuttle computers," IBM Journal of Research and Development, vol. 20, no. 1, January 1976, 20-28.

STIF 76 J. J. Stiffler, "Architectural design for near-100% fault coverage," Proc. 1976 IEEE International Symposium on Fault-Tolerant Computing, June 21-23, 1976, Pittsburgh, PA.

SZYG 76 S. A. Szygenda and E. W. Thompson, "Modeling and digital simulation for design verification and diagnosis," IEEE Transactions on Computers, vol. C-25, no. 12, December 1976, 1242-1253.

TAND 76 Tandem 16 System Description, Tandem Computers Inc., Cupertino, CA, October 1976.

TANG 69 D. T. Tang and R. T. Chien, "Coding for error control," IBM Systems Journal, vol. 8, no. 1, 1969, 48-86.

TAYL 73 P. S. Tayler, "A reliability and comparative analysis of two standby system configurations," IEEE Transactions on Reliability, vol. R-22, April 1973, pp. 13-15.

TOYW 78 W. N. Toy, "Fault-tolerant design of local ESS processors," Proc. IEEE, vol. 66, no. 10, October 1978, pp. 1126-1145.

ULTR 74 Reconfigurable computer system study, Ultrasystems, Inc., Newport Beach, CA, performed for NASA Langley Research Center, 1974.

VONN 56 J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," Automata Studies, C. E. Shannon and J. McCarthy, eds., Ann. of Math. Studies No. 34, Princeton University Press, 1956, 43-98.

WAKE 74 J. F. Wakerly and E. J. McCluskey, "Design of low-cost general-purpose self-diagnosing computers," Information Processing 74, North-Holland Publ. Co., Amsterdam, 1974, 108-111.

WENS 72 J. H. Wensley, "SIFT - software implemented fault-tolerance," AFIPS Conf. Proc. vol. 41, part 1, 1972, 243-254.

WENS 76 J. H. Wensley et al., "The design, analysis and verification of the SIFT fault-tolerant system," Proc. Second Int. Conference on Software Engineering, San Francisco, CA, October 1976, 458-469.

WENS 78 J. H. Wensley, et al., "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," Proc. IEEE, vol. 66, no. 10, October 1978, pp. 1240-1255.

WYLE 67 Wyle, H. and G. Burnett, "Some relationships between failure detection probability and computer system reliability," AFIPS Conference Proceedings, Fall Joint Computer Conference, 1967, 745-756.

Supplement

KILP 72 P. S. Kilpatrick et al., All semiconductor distributed aerospace processor/memory study, Volume 1 Avionics Processing Requirements, Honeywell, Inc., AFAL TR-72, performed for AFAC, Wright-Patterson Air Force Base, Ohio, 1972.

1553A Aircraft Internal Time Division Command/Response Multiplex Data Bus, DoD Military Standard 1553A, April 1975, U. S. Government Printing Office 1975-603-767-/1472.

APPENDIX

I/O BUILDING BLOCKS

I/O BUILDING BLOCKS

Input-output requirements of host systems vary widely in voltage ranges, currents, and timing parameters. The approach best suited to building-block development is to provide a standard set of functions which serve a majority of general applications. The user is required to supply any additional special functions unique to his applications.

To be consistent with the FTBBC computer modules, all building blocks must provide memory-mapped I/O. This is, each I/O building block must recognize its identification and the function being requested from an out-of-range address appearing on the host computer's address bus. Data for output or input is transferred over the data bus in response to a write or read to the specified I/O address.

A second set of requirements is related to fault-tolerance. The I/O building block must check incoming addresses and data for proper coding, and utilize duplication or coding checks to verify proper functioning of its internal logic. Either an error in incoming data or detection of an internal fault must signal an error indication to the CORE Building Block. This internal error indicator should be a morphic (one-out-of-two) coded signal to prevent a single point failure. Finally, the building block must encode incoming data for presentation on the host computer's bus.

Typical I/O Functions

The following is a listing of I/O functions which should be supplied by building-block modules. One special feature is important in achieving synchronization in voting configurations, as well as decoupling I/O timing from detailed instruction timing in the Terminal module. This is a feature which creates a granularity in I/O timing by synchronizing outputs and inputs with the Real-time interrupt which drives the computer system. Specifically, a Real Time Interrupt (RTI) input is provided with each building block, and which typically provides a pulse every few milliseconds. All output commands are held within the building block, and are executed at precisely the next RTI. Similarly, inputs are sampled and held through an RTI period.

When a synchronous executive is employed in the Terminal Module, this technique allows software to be changed without changing the I/O timing of unmodified programs [REDW 78b]. It also prevents DMA activity from the intercommunications bus from changing I/O timing due to slight variations in processor speed due to stolen memory cycles. Finally a restricted interaction with the host system coupled with synchronous software operation is expected to simplify verification and validation.

(1) I/O Function #1 Parallel Data Out. Outputs a 16-bit data word taken from the host computer's data bus at the next RTI pulse.

(2) I/O Function #2 Parallel Data Input. Sample and hold a 16-bit data word at the next RTI pulse. A separate Read Command transfers the sampled data into the host computer.

(3) I/O Function #3 Serial Data Out. Shifts out a 16-bit data word at the next RTI pulse. Provides word gate and shift clock signals.

(4) I/O Function #4 Serial Data In. Accepts up to 16 bits of serial digital data from a data source.

(5) I/O Function #5 Pulse and Bilevel Input. This function is used to sense the logical state of up to 8 lines, and to sense the occurrence of a pulse event within a software determined measurement period. The pulse sensing logic is reset on RTI (or multiples of RTI) time centers while the level sense logic is allowed to change state on 1 μsec intervals.

(6) I/O Function #6 Pulse Counter. This function totals the number of pulse events over a predetermined time interval.

(7) I/O Function #7 Adjustable Frequency Generator (Modulo N Counter). Used to generate pulse streams which are integral submultiples of a Master Clock.

(8) I/O Function #8 Pulse Output. Generates pulses with delay and width program-specified and derived from a Master Clock. Pulses are generated periodically on RTI time centers and are typically of 10μ sec or 100μ sec duration.

(9) I/O Function #9 Analog Multiplexor. Up to 16 lines of analog data can be collected, in a two-part operation. First the desired analog line is selected and the data is quantized at the next RTI time. The resulting digital data is held in an output register until it is retrieved by software with a subsequent read operation.

(10) I/O Function #10 High Rate DMA. This function is designed to minimize handling of high rate data. A starting address and word count is loaded into the building block along with an output or receive request. Data is transferred to or from the host computer memory under the control of a peripheral device.

These functions were selected to provide a general I/O capability. The Functions are made sufficiently powerful so that the burden of high rate timing can be removed from software. In general, the software only has to provide outputs with a resolution of a few milliseconds (determined by the RTI) and the hardware takes care of the finer details.

In order to proceed with I/O building block design, a detailed analysis of NAVY systems and procedures is required. However, the following general comments can be made regarding building block implementation.

Implementation Strategy

The circuitry for each I/O function is not complex and the implementation of fault-detection is straightforward. Where information structure is preserved (such as data in and out) parity checking can be employed. Control functions can be duplicated with morphic comparison.

The density of VLSI technology is sufficiently high that a number of I/O functions can be placed on a single chip. The specific function which is required can be activated by connecting pins. This technique can reduce the inventory of building blocks to two or three. Most of the functions described above can be implemented on a single chip.

One additional requirement is for the redundant use of I/O elements. To achieve redundancy in Terminal Modules, two or more modules are cross-strapped, i.e., their inputs and outputs are hooked together. One module is powered and the others are used as unpowered standby spares. When cross-strapped I/O is used, it is important that short-protection be provided at all output connections. Otherwise a shorted I/O connection could inactivate all of the spares. Typical techniques for protection are to isolate outputs with series diodes and inputs with series resistors. Thus, hybrid isolator packages will be required as an integral part or as an adjunct to the building blocks.

I/O Building Block definition is an area recommended for further study in the areas of (1) a detailed definition of NAVY functional requirements and (2) chip development.